

---

# Distributed SDDM Solvers: Theory & Applications

---

**Rasul Tutunov**

Department of Computer and Information Science,  
University of Pennsylvania,  
tutunov@seas.upenn.edu

**Haitham Bou-Ammar**

Department of Computer and Information Science,  
University of Pennsylvania,  
Department of Operations Research and Financial Engineering,  
Princeton University,  
haithamb@seas.upenn.edu

**Ali Jadbabaie,**

Department of Electrical and Systems Engineering,  
University of Pennsylvania,  
jadbabaie@seas.upenn.edu

## Abstract

In this paper, we propose distributed solvers for systems of linear equations given by symmetric diagonally dominant M-matrices based on the parallel solver of Spielman and Peng. We propose two versions of the solvers, where in the first, full communication in the network is required, while in the second communication is restricted to the R-Hop neighborhood between nodes for some  $R \geq 1$ . We rigorously analyze the convergence and convergence rates of our solvers, showing that our methods are capable of outperforming state-of-the-art techniques.

Having developed such solvers, we then contribute by proposing an accurate distributed Newton method for network flow optimization. Exploiting the sparsity pattern of the dual Hessian, we propose a Newton method for network flow optimization that is both faster and more accurate than state-of-the-art techniques. Our method utilizes the distributed SDDM solvers for determining the Newton direction up to any arbitrary precision  $\epsilon > 0$ . We analyze the properties of our algorithm and show superlinear convergence within a neighborhood of the optimal. Finally, in a set of experiments conducted on randomly generated and barbell networks, we demonstrate that our approach is capable of significantly outperforming state-of-the-art techniques.

## 1 Introduction

Solving systems of linear equations given by symmetric diagonally matrices (SDD) is of interest to researchers in a variety of fields. Such constructs, for example, are used to determine solutions to partial differential equations [18] and computations of maximum flows in graphs [19, 20]. Other application domains include machine learning [21, 22], and computer vision [34]<sup>1</sup>.

---

<sup>1</sup>This research is supported in parts by by ONR grant Number N00014-12-1-0997 and AFOSR grant FA9550-13-1-0097.

Much interest has been devoted to determining fast algorithms for solving SDD systems. Recently, Spielman and Teng [24], utilized the multi-level framework of [25], pre-conditioners [26], and spectral graph sparsifiers [27], to propose a nearly linear-time algorithm for solving SDD systems. Further exploiting these ingredients, Koutis *et. al* [28, 29] developed an even faster algorithm for acquiring  $\epsilon$ -close solutions to SDD linear systems. Further improvements have been discovered by Kelner *et. al* [30], where their algorithm relied on only spanning-trees and eliminated the need for graph sparsifiers and the multi-level framework.

Motivated by applications, much progress has been made in developing parallel versions of these algorithms. Koutis and Miller [31] proposed an algorithm requiring nearly-linear work (i.e., total number of operations executed by a computation) and  $m^{1/6}$  depth (i.e., longest chain of sequential dependencies in the computation) for planar graphs. This was then extended to general graphs in [32] leading to depth close to  $m^{1/3}$ . Since then, Peng and Spielman [11] have proposed an efficient parallel solver requiring nearly-linear work and poly-logarithmic depth without the need for low-stretch spanning trees. Their algorithm, which we provide a distributed construction for, requires sparse approximate inverse chains [11] which facilitates the solution of the SDD system.

Less progress, on the other hand, has been made on the distributed version of these solvers. Contrary to the parallel setting, memory is not shared and is rather distributed in the sense that each unit abides by its own memory restrictions. Furthermore, communication in a distributed setting fundamentally relies on message passing through communication links. Current methods, e.g., Jacobi iteration [16, 17], can be used for such distributed solutions but require substantial complexity. In [33], the authors propose a gossiping framework for acquiring a solution to SDDM systems in a distributed fashion. Recent work [34] considers a local and asynchronous solution for solving systems of linear equations, where they acquire a bound on the number of needed multiplication proportional to the degree and condition number of the graph for one component of the solution vector.

**Contributions:** In this paper, we propose a fast distributed solver for linear equations given by symmetric diagonally dominant M-Matrices. Our approach distributes the parallel solver in [11] by considering a specific approximated inverse chain which can be computed efficiently in a distributed fashion. We develop two versions of the solver. The first, requires full communication in the network, while the second is restricted to R-Hop neighborhood of nodes for some  $R \geq 1$ . Similar to the work in [11], our algorithms operate in two phases. In the first, a “crude” solution to the system of equations is returned, while in the second a distributed R-Hop restricted pre-conditioner is proposed to drive the “crude” solution to an  $\epsilon$ -approximate one for any  $\epsilon > 0$ . Due to the distributed nature of the setting considered, the direct application of the sparsifier and pre-conditioner of Peng and Spielman [11] is difficult due to the need of global information. Consequently, we propose a new sparse inverse chain which can be computed in a decentralized fashion for determining the solution to the SDDM system.

Interestingly, due to the involvement of powers of matrices with eigenvalues less than one, our inverse chain is substantially shorter compared to that in [11]. This leads us to a distributed SDDM solver with lower computational complexity compared to state-of-the-art methods. Specifically, our algorithm’s complexity is given by

$$\mathcal{O}\left(n^3 \frac{\alpha}{R} \frac{W_{\max}}{W_{\min}} \log\left(\frac{1}{\epsilon}\right)\right),$$

with  $n$  being the number of nodes in graph  $\mathcal{G}$ ,  $W_{\max}$  and  $W_{\min}$  denoting the largest and smaller weights of the edges in  $\mathcal{G}$ , respectively,  $\alpha = \min\left\{n, \frac{d_{\max}^{R+1}-1}{d_{\max}-1}\right\}$  representing the upper bound on the size of the R-Hop neighborhood  $\forall v \in \mathcal{V}$ , and  $\epsilon \in (0, \frac{1}{2}]$  being the precision parameter. Furthermore, our approach improves current linear methods by a factor of  $\log n$  and by a factor of the degree compared to [34] for each component of the solution vector.

Having developed such distributed solvers, we next contribute by proposing an accurate distributed Newton method for network flow optimization. Exploiting the sparsity pattern of the dual Hessian, we propose a Newton method for network optimization that is both faster and more accurate than state-of-the-art techniques. Our method utilizes the proposed SDDM distributed solvers to approximate the Newton direction up to any arbitrary  $\epsilon > 0$ . The resulting algorithm is an efficient and accurate distributed second-order method which performs almost identically to exact Newton. We

analyze the properties of the proposed algorithm and show that, similar to conventional Newton methods, superlinear convergence within a neighborhood of the optimal value is attained. We finally demonstrate the effectiveness of the approach in a set of experiments on randomly generated and Barbell networks.

## 2 The parallel SDDM Solver

We now review the parallel solver for symmetric diagonally dominant (SDD) linear systems [11].

### 2.1 Problem Setting

As detailed in [11], SDDM solvers consider the following system of linear equations:

$$\mathbf{M}_0 \mathbf{x} = \mathbf{b}_0 \quad (1)$$

where  $\mathbf{M}_0$  is a Symmetric Diagonally Dominant M-Matrix (SDDM). Namely,  $\mathbf{M}_0$  is symmetric positive definite with non-positive off diagonal elements, such that for all  $i = 1, 2, \dots, n$ :

$$[\mathbf{M}_0]_{ii} \geq - \sum_{j=1, j \neq i}^n [\mathbf{M}_0]_{ij}.$$

The system of Equations in 1 can be interpreted as representing an undirected weighted graph,  $\mathcal{G}$ , with  $\mathbf{M}_0$  being its Laplacian. Namely,  $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathbf{W})$ , with  $\mathcal{V}$  representing the set of nodes,  $\mathcal{E}$  denoting the edges, and  $\mathbf{W}$  representing the weighted graph adjacency. Nodes  $v_i$  and  $v_j$  are connected with an edge  $e = (i, j)$  iff  $\mathbf{W}_{ij} > 0$ , where:

$$\mathbf{W}_{ij} = -[\mathbf{M}_0]_{ij}.$$

Following [11], we seek  $\epsilon$ -approximate solutions to  $\mathbf{x}^*$ , being the exact solution of  $\mathbf{M}_0 \mathbf{x} = \mathbf{b}_0$ , defined as:

**$\epsilon$ -Approximate Solution** Let  $\mathbf{x}^* \in \mathbb{R}^n$  be the solution of  $\mathbf{M}_0 \mathbf{x} = \mathbf{b}_0$ . A vector  $\tilde{\mathbf{x}} \in \mathbb{R}^n$  is called an  $\epsilon$ -approximate solution, if:

$$\|\mathbf{x}^* - \tilde{\mathbf{x}}\|_{\mathbf{M}_0} \leq \epsilon \|\mathbf{x}^*\|_{\mathbf{M}_0}, \quad \text{where } \|\mathbf{u}\|_{\mathbf{M}_0}^2 = \mathbf{u}^\top \mathbf{M}_0 \mathbf{u}. \quad (2)$$

The R-hop neighbourhood of node  $v_k$  is defined as  $\mathbb{N}_R(v_k) = \{v \in \mathcal{V} : \text{dist}(v_k, v) \leq R\}$ . We also make use of the diameter of a graph,  $\mathcal{G}$ , defined as  $\text{diam}(\mathcal{G}) = \max_{v_i, v_j \in \mathcal{V}} \text{dist}(v_i, v_j)$ .

**Sparsity Pattern** We say that a matrix  $\mathbf{A} \in \mathbb{R}^{n \times n}$  has a sparsity pattern corresponding to the R-hop neighborhood if  $\mathbf{A}_{ij} = 0$  for all  $i = 1, \dots, n$  and for all  $j$  such that  $v_j \notin \mathbb{N}_R(v_i)$ .

We will denote the spectral radius of a matrix  $\mathbf{A}$  by  $\rho(\mathbf{A}) = \max |\lambda_i|$ , where  $\lambda_i$  represents an eigenvalue of the matrix  $\mathbf{A}$ . Furthermore, we will make use of the condition number<sup>2</sup>,  $\kappa(\mathbf{A})$  of a matrix  $\mathbf{A}$  defined as  $\kappa = \left| \frac{\lambda_{\max}(\mathbf{A})}{\lambda_{\min}(\mathbf{A})} \right|$ . In [11] it is shown that the condition number of the graph Laplacian is at most

$$\mathcal{O} \left( n^3 \frac{\mathbf{W}_{\max}}{\mathbf{W}_{\min}} \right),$$

where  $\mathbf{W}_{\max}$  and  $\mathbf{W}_{\min}$  represent the largest and the smallest edge weights in  $\mathcal{G}$ . Finally, the condition number of a sub-matrix of the Laplacian is at most  $\mathcal{O} \left( n^4 \frac{\mathbf{W}_{\max}}{\mathbf{W}_{\min}} \right)$ , see [11].

### 2.2 Standard Splittings & Approximations

Our first contribution is a distributed version of the parallel solver for SDDM systems of equations previously proposed in [11]. Before detailing our solver, however, we next introduce basic mathematical machinery needed for developing the parallel solver of [11]. The parallel solver commences by considering the standard splitting of the symmetric matrix  $\mathbf{M}_0$ :

<sup>2</sup>Please note that in the case of the graph Laplacian, the condition number is defined as the ratio of the largest to the smallest nonzero eigenvalues.

**Definition** The standard splitting of a symmetric matrix  $M_0$  is:

$$M_0 = D_0 - A_0. \quad (3)$$

Here,  $D_0$  is a diagonal matrix consisting of the diagonal elements in  $M_0$  such that:

$$[D_0]_{ii} = [M_0]_{ii} \quad \forall i = 1, 2, \dots, n.$$

Furthermore,  $A_0$  is a non-negative symmetric matrix such that:

$$[A_0]_{ij} = \begin{cases} -[M_0]_{ij} & : \text{if } i \neq j, \\ 0 & : \text{otherwise.} \end{cases}$$

To quantify the quality of the acquired solutions, we define two additional mathematical constructs. First, the Loewner ordering is defined as:

**Definition** Let  $\mathcal{S}_{(n)}$  be the space of  $n \times n$ -symmetric matrices. The Loewner ordering  $\preceq$  is a partial order on  $\mathcal{S}_{(n)}$  such that  $Y \preceq X$  if and only if  $X - Y$  is positive semidefinite.

Having defined the Loewner order, we next define the notion of approximation for matrices “ $\approx_\alpha$ ”:

**Definition** Let  $X$  and  $Y$  be positive semidefinite symmetric matrices. Then  $X \approx_\alpha Y$  if and only iff

$$e^{-\alpha} X \preceq Y \preceq e^{\alpha} X \quad (4)$$

with  $A \preceq B$  meaning  $B - A$  is positive semidefinite.

Based on the above definitions, the following lemma represents the basic characteristics of the  $\approx_\alpha$  operator:

**Lemma 1.** [11] Let  $X, Y, Z$  and  $Q$  be symmetric positive semi definite matrices. Then

- (1) If  $X \approx_\alpha Y$ , then  $X + Z \approx_\alpha Y + Z$ ,
- (2) If  $X \approx_\alpha Y$  and  $Z \approx_\alpha Q$ , then  $X + Z \approx_\alpha Y + Q$ ,
- (3) If  $X \approx_{\alpha_1} Y$  and  $Y \approx_{\alpha_2} Z$ , then  $X \approx_{\alpha_1 + \alpha_2} Z$
- (4) If  $X$ , and  $Y$  are non singular and  $X \approx_\alpha Y$ , then  $X^{-1} \approx_\alpha Y^{-1}$ ,
- (5) If  $X \approx_\alpha Y$  and  $V$  is a matrix, then  $V^T X V \approx_\alpha V^T Y V$ .

Since the parallel solver returns an approximation,  $Z_0$ , to  $M_0^{-1}$  (see Section 2), the following lemma shows that “good” approximations to  $M_0^{-1}$  guarantee “good” approximate solutions to  $M_0 x = b_0$ .

**Lemma 2.** Let  $Z_0 \approx_\epsilon M_0^{-1}$ , and  $\tilde{x} = Z_0 b_0$ , then  $\tilde{x}$  is  $\sqrt{2^\epsilon(e^\epsilon - 1)}$  approximate solution to  $M_0 x = b_0$ .

*Proof.* The proof can be found in the appendix. □

### 2.3 The Solver

The parallel SDDM solver proposed in [11] is a parallelized technique for solving the problem of Section 2.1. It makes use of inverse approximated chains (see Definition 2.3) to determine  $\tilde{x}$  and can be split in two steps. In the first, Algorithm 1, a “crude” approximation,  $x_0$ , of  $\tilde{x}$  is returned.  $x_0$  is driven to the  $\epsilon$ -close solution,  $\tilde{x}$ , using Richardson Preconditioning in Algorithm 2. Before we proceed, we start with the following two Lemmas which enable the definition of inverse chain approximation.

**Lemma 3.** [11] If  $M = D - A$  is an SDDM matrix, with  $D$  being positive diagonal, and  $A$  denoting a non-negative symmetric matrix, then  $D - AD^{-1}A$  is also SDDM.

**Lemma 4.** [11] Let  $M = D - A$  be an SDDM matrix, where  $D$  is positive diagonal and  $A$  a symmetric matrix. Then

$$(D - A)^{-1} = \frac{1}{2} \left[ D^{-1} + (I + D^{-1}A) (D - AD^{-1}A)^{-1} (I + AD^{-1}) \right]. \quad (5)$$

Given the results in Lemmas 3 and 4, we now can consider inverse approximated chains of  $M_0$ :

**Definition** Let  $\mathcal{C} = \{M_0, M_1, \dots, M_d\}$  be a collection of SDDM matrices such that  $M_i = D_i - A_i$ , with  $D_i$  a positive diagonal matrix, and  $A_i$  denoting a non-negative symmetric matrix. Then  $\mathcal{C}$  is an inverse approximated chain if there exists positive real numbers  $\epsilon_0, \epsilon_1, \dots, \epsilon_d$  such that:

- (1)  $D_i - A_i \approx_{\epsilon_{i-1}} D_{i-1} - A_{i-1} D_{i-1}^{-1} A_{i-1} \quad \forall i = 1, \dots, d$ ,
- (2)  $D_i \approx_{\epsilon_{i-1}} D_{i-1}$ , and
- (3)  $D_d \approx_{\epsilon_d} D_d - A_d$ .

It is shown in [11] that an approximate inverse chain allows for “crude” solutions to the system of linear equations in  $D_0 - A_0$  in time proportional to the number of non-zeros entries in the matrices in the inverse chain. Such a procedure is summarized in the following algorithm:

---

**Algorithm 1** ParallelRSolve ( $M_0, M_1, \dots, M_d, b_0$ )

---

```

1: Input: Inverse approximated chain,  $\{M_0, M_1, \dots, M_d\}$ , and  $b_0$  being
2: Output: The “crude” approximation,  $x_0$ , of  $x^*$ 
3: for  $i = 1$  to  $d$  do
4:    $b_i = (I + A_{i-1} D_{i-1}^{-1}) b_{i-1}$ 
5: end for
6:  $x_d = D_d^{-1} b_d$ 
7: for  $i = d - 1$  to  $0$  do
8:    $x_i = \frac{1}{2} [D_i^{-1} b_i + (I + D_i^{-1} A_i) x_{i+1}]$ 
9: end for
10: return  $x_0$ 

```

---

On a high level, Algorithm 2.3 operates in two phases. In the first (i.e., lines 3-5) a forward loop (up-to the length of the inverse chain  $d$ ) computes intermediate vectors  $b_i$  as:

$$b_i = (I + A_{i-1} D_{i-1}^{-1}) b_{i-1}, \quad (6)$$

for  $i = \{1, \dots, d\}$ . These can then be used to compute the “crude” solution  $x_0$  using a “backward” loop (i.e., lines 7-9). Consequently, the crude solution is computed iteratively backwards as:

$$x_i = \frac{1}{2} [D_i^{-1} b_i + (I + D_i^{-1} A_i) x_{i+1}],$$

with  $x_d = D_d^{-1} b_d$  and  $b_i$  as defined in Equation 6. The quality of the “crude” solution returned by the Algorithm is quantified in the following lemma:

**Lemma 5.** [11] Let  $\{M_0, M_1, \dots, M_d\}$  be the inverse approximated chain and denote  $Z_0$  be the operator defined by ParallelRSolve ( $M_0, M_1, \dots, M_d, b_0$ ), namely,  $x_0 = Z_0 b_0$ . Then

$$Z_0 \approx_{\sum_{i=0}^d \epsilon_i} M_0^{-1} \quad (7)$$

Having returned a “crude” solution to  $M_0 x = b_0$ , the authors in [11] obtain arbitrary close solutions using the *preconditioned Richardson iterative scheme*. The first step in the exact solver is the usage of Algorithm 2.3 to obtain the “crude” solution  $\chi$ . This is then updated through the loop in lines 4-8 to obtain an  $\epsilon$ -close solution to  $x^*$ , see Algorithm 2. Following the analysis in [11], Lemma 6 provides the iteration count needed by Algorithm 2 to arrive at  $\tilde{x}$ :

**Lemma 6.** [11] Let  $\{M_0, M_1, \dots, M_d\}$  be an inverse approximated chain such that  $\sum_{i=1}^d \epsilon_i < \frac{1}{3} \ln 2$ . Then ParallelESolve ( $M_0, M_1, \dots, M_d, b_0, \epsilon$ ) requires  $q$  iterations to arrive at an  $\epsilon$  close solution of  $x^*$  with:  $q = \mathcal{O}(\log \frac{1}{\epsilon})$ .

---

**Algorithm 2** ParallelESolve ( $M_0, M_1, \dots, M_d, b_0, \epsilon$ )

---

1: **Input:** Inverse approximated chain  $\{M_0, M_1, \dots, M_d\}$ ,  $b_0$ , and  $\epsilon$ .  
2: **Output:**  $\epsilon$  close approximation,  $\tilde{x}$ , of  $x^*$   
3: **Initialize:**  $y_0 = 0$ ;  
     $\chi = \text{ParallelRSolve}(M_0, M_1, \dots, M_d, b_0)$  (i.e., Algorithm 1)  
4: **for**  $k = 1$  to  $q$  **do**  
5:    $u_k^{(1)} = M_0 y_{k-1}$   
6:    $u_k^{(2)} = \text{ParallelRSolve}(M_0, M_1, \dots, M_d, u_k^{(1)})$   
7:    $y_k = y_{k-1} - u_k^{(2)} + \chi$   
8: **end for**  
9:  $\tilde{x} = y_q$   
10: **return**  $\tilde{x}$

---

### 3 Distributed SDDM Solvers

Having introduced the parallel solver, next we detail our first contribution by proposing a distributed solver for SDDM linear systems. In particular, we develop two versions. The first, requires full communication in the network, while the second restricts communication to the R-Hop neighborhood increasing its applicability. Not only our solver improves the computational complexity of distributed methods for system of equations represented by an SDDM matrix, but can also be applied to a variety of fields including distributed Newton methods, computer vision, among others.

To compute the solution of SDDM systems in a distributed fashion, we follow a similar strategy to that of [11] with major differences. Our distributed solver requires two steps to arrive at an  $\epsilon$ -close approximation to  $x^*$ . Similar to [11], the first step adopts an inverse approximated chain to determine a “crude” solution to  $x^*$ . The inverse chain proposed in [11] can not be computed in a distributed fashion rendering its immediate application to our setting difficult. Hence, we par-ways with [11] by proposing an inverse chain which can be computed in a distributed fashion. This chain, defined in Section 3.1.1, enables the distributed computation of both a crude and exact solution to  $M_0 x = b_0$ . Interestingly, due to the involvement of matrices with eigenvalues less than 1, the length,  $d$ , of our inverse chain is substantially shorter compared to that of [11], allowing for fast and efficient distributed solvers. Given the crude solution, the second step computes an  $\epsilon$ -close approximation to  $x^*$ . This is achieved by proposing a distributed version of the Richardson pre-conditioning scheme. Definitely, this step is also similar in spirit to that in [11], but generalizes the aforementioned authors’ work into a distributed setting and allows for  $\epsilon$ -close approximation to  $x^*$  for *any arbitrary*  $\epsilon > 0$ . Main results on the full communication version of the solver are summarized in the following theorem:

**Theorem 1.** *There exists a distributed algorithm,*

$$\mathcal{A}(\{[M_0]_{k1}, \dots [M_0]_{kn}\}, [b_0]_k, \epsilon),$$

*that computes  $\epsilon$ -close approximations to the solution of  $M_0 x = b_0$  in  $\mathcal{O}(n^2 \log \kappa \log(\frac{1}{\epsilon}))$  time steps, with  $n$  the number of nodes in  $\mathcal{G}$ ,  $\kappa$  the condition number of  $M_0$ , and  $[M_0]_k$  the  $k^{th}$  row of  $M_0$ , as well as  $\epsilon \in (0, \frac{1}{2}]$  representing the precision parameter.*

The above distributed algorithms require no knowledge of the graph’s topology, but do require the information from all other nodes (i.e., full communication) for computing solutions to  $M_0 x = b_0$ . In a variety of real-world applications (e.g., smart-grids, transportation) load, capacity, money and resource restrictions pose problems for such a requirement. Consequently, we extend the previous solvers to an R-Hop version in which communication is restricted to the R-Hop neighborhood between nodes for some  $R \geq 1$ . Again we follow a two-step strategy, where in the first we compute the crude solution and in the second an  $\epsilon$ -close approximation to  $x_0$  is determined using an “R-Hop restricted” Richardson pre-conditioner. These results are captured in the following theorem:

**Theorem 2.** *There is a decentralized algorithm,*

$$\mathcal{A}(\{[M_0]_{k1}, \dots [M_0]_{kn}\}, [b_0]_k, R, \epsilon),$$

that uses only  $R$ -Hop communication between the nodes and computes  $\epsilon$ -close solutions to  $M_0x = b_0$  in

$$\mathcal{O}\left(\left(\frac{\alpha\kappa}{R} + \alpha R d_{max}\right) \log\left(\frac{1}{\epsilon}\right)\right)$$

time steps, with  $n$  being the number of nodes in  $\mathcal{G}$ ,  $d_{max}$  denoting the maximal degree,  $\kappa$  the condition number of  $M_0$ , and  $\alpha = \min\left\{n, \frac{(d_{max}^{R+1}-1)}{(d_{max}-1)}\right\}$  representing the upper bound on the size of the  $R$ -hop neighborhood  $\forall v \in \mathcal{V}$ , and  $\epsilon \in (0, \frac{1}{2}]$  being the precision parameter.

The remainder of the section details the above distributed solvers and provides rigorous theoretical guarantees on the convergence and convergence rates of each of the algorithms. We start by describing solvers requiring full network communication and then detail the  $R$ -Hop restricted versions.

### 3.1 Full Communication Distributed Solvers

As mentioned previously, our strategy for a distributed implementation of the parallel solver in [11] requires two steps. In the first a “crude” solution is returned, while in the second an  $\epsilon$ -close approximation (for any arbitrary  $\epsilon > 0$ ) to  $x_0$  is computed.

#### 3.1.1 “Crude” Distributed SDDM Solvers

The distributed crude solver, represented in Algorithm 3, resembles similarities to the parallel one of [11] with major differences. On a high level, Algorithm 3 operates in two distributed phases. In the first, a forward loop computes intermediate  $b$  vectors which are then used to update the crude solution of  $M_0x = b_0$ . The crucial difference to [11], however, is the distributed nature of these computations. Precisely, the algorithm is responsible for determining the crude solution for each node  $v_k \in \mathcal{V}$ . Due to such distributed nature, the inverse approximated chain used in [11] is inapplicable to our setting. Therefore, the second crucial difference to the parallel SDDM solver is the introduction of a new chain which can be computed in a distributed fashion. Starting from  $M_0 = D_0 - A_0$ , our “crude” distributed solver makes use of the following collection as the inverse approximated chain:

$$\mathcal{C} = \{A_0, D_0, A_1, D_1, \dots, A_d, D_d\}, \quad (8)$$

where  $D_k = D_0$ , and  $A_k = D_0 (D_0^{-1} A_0)^{2^k}$ , for  $k = \{1, \dots, d\}$  with  $d$  being the length of the inverse chain. Note that since the magnitude of the eigenvalues of  $D_0^{-1} A_0$  is strictly less than 1,  $(D_0^{-1} A_0)^{2^k}$  tends to zero as  $k$  increases which reduces the length of the chain needed. This length is explicitly computed in Section 3.1.3 for attaining  $\epsilon$ -close approximations to  $x^*$ .

It is relatively easy to verify that  $\mathcal{C}$  is an inverse approximated chain, since:

- (1)  $D_i - A_i \approx_{\epsilon_{i-1}} D_{i-1} - A_{i-1} D_{i-1}^{-1} A_{i-1}$  with  $\epsilon_i = 0$  for  $i = 1, \dots, d$ ,
- (2)  $D_i \approx_{\epsilon_{i-1}} D_{i-1}$  with  $\epsilon_i = 0$  for  $i = 1, \dots, d$ , and
- (3)  $D_d \approx_{\epsilon_d} D_d - A_d$ .

Algorithm 3 returns the  $k^{th}$  component of the approximate solution vector,  $[x_0]_k$ . As inputs it requires the inverse chain of Equation 8, the  $k^{th}$  component of  $b_0$ , and the length of the inverse chain. Namely, each node,  $v_k \in \mathcal{V}$ , receives the  $k^{th}$  row of  $M_0$ , the  $k^{th}$  value of  $b_0$  (i.e.,  $[b_0]_k$ ), and the length of the inverse approximated chain  $d$  and then operates in two parts. In the first (i.e., lines 1-8) a forward loop computes the  $k^{th}$  component of  $b$  exploiting the distributed inverse chain, while in the second a backward loop (lines 9-17) is responsible for computing the  $k^{th}$  component of the “crude” solution  $[x_0]_k$  which is then returned. Essentially, in both the forward and backward loops each of the  $b$  and  $x$  vectors are computed in a distributed fashion based on the relevant components of the matrices, explaining the usage of  $\mathbb{N}$  loops in Algorithm 3.

**Theoretical Guarantees of Algorithm 3:** Due to the modifications made to the original parallel solver, new theoretical analysis quantifying convergence and accuracy of the returned “crude” solu-

---

**Algorithm 3** DistrRSolve $\left(\{[M_0]_{k1}, \dots, [M_0]_{kn}\}, [b_0]_k, d\right)$ 


---

```

1: Part One: Computing  $[b_i]_k$ 
2:  $[b_1]_k = [b_0]_k + \sum_{j:v_j \in \mathbb{N}_1(v_k)} [A_0 D_0^{-1}]_{kj} [b_0]_j$ 
3: for  $i = 2$  to  $d$  do
4:   for  $j : v_j \in \mathbb{N}_{2^{i-1}}(v_k)$  do
5:      $\left[(A_0 D_0^{-1})^{2^{i-1}}\right]_{kj} = \sum_{r=1}^n \frac{[D_0]_{rr}}{[D_0]_{jj}} \left[(A_0 D_0^{-1})^{2^{i-2}}\right]_{kr} \left[(A_0 D_0^{-1})^{2^{i-2}}\right]_{jr}$ 
6:   end for
7:    $[b_i]_k = [b_{i-1}]_k + \sum_{j:v_j \in \mathbb{N}_{2^{i-1}}(v_k)} \left[(A_0 D_0^{-1})^{2^{i-1}}\right]_{kj} [b_{i-1}]_j$ 
8: end for

9: Part Two: Computing  $[x_0]_k$ 
10:  $[x_d]_k = [b_d]_k / [D_0]_{kk}$ 
11: for  $i = d - 1$  to  $1$  do
12:   for  $j : v_j \in \mathbb{N}_{2^i}(v_k)$  do
13:      $\left[(D_0^{-1} A_0)^{2^i}\right]_{kj} = \sum_{r=1}^n \frac{[D_0]_{jj}}{[D_0]_{rr}} \left[(D_0^{-1} A_0)^{2^{i-1}}\right]_{kr} \left[(D_0^{-1} A_0)^{2^{i-1}}\right]_{jr}$ 
14:   end for
15:    $[x_i]_k = \frac{[b_i]_k}{2[D_0]_{kk}} + \frac{[x_{i+1}]_{k+1}}{2} + \frac{1}{2} \sum_{j:v_j \in \mathbb{N}_{2^i}(v_k)} \left[(D_0^{-1} A_0)^{2^i}\right]_{kj} [x_{i+1}]_j$ 
16: end for
17:  $[x_0]_k = \frac{[b_0]_k}{2[D_0]_{kk}} + \frac{[x_1]_k}{2} + \frac{1}{2} \sum_{j:v_j \in \mathbb{N}_1(v_k)} [D_0^{-1} A_0]_{kj} [x_1]_j$ 
18: return:  $[x_0]_k$ 

```

---

tion is needed. We show that DistrRSolve computes the  $k^{th}$  component of the “crude” approximation of  $x^*$  and provide time complexity analysis. These results are summarized in the following lemma<sup>3</sup>:

**Lemma 7.** *Let  $M_0 = D_0 - A_0$  be the standard splitting of  $M_0$ . Let  $Z'_0$  be the operator defined by DistrRSolve $(\{[M_0]_{k1}, \dots, [M_0]_{kn}\}, [b_0]_k, d)$  (i.e.,  $x_0 = Z'_0 b_0$ ). Then*

$$Z'_0 \approx_{\epsilon_d} M_0^{-1}.$$

Moreover, Algorithm 3 requires  $\mathcal{O}(dn^2)$  time steps.

In words, Lemma 7 states that Algorithm 3 requires  $\mathcal{O}(dn^2)$  to arrive at an  $\epsilon_d$  approximation to the real inverse  $M_0^{-1}$ , where this approximation is quantified using Definition 2.2:

$$e^{-\epsilon_d} Z'_0 \preceq M_0^{-1} \preceq e^{\epsilon_d} Z'_0.$$

$Z'_0$  can then be used to compute the crude solution as  $x_0 = Z'_0 b$ . Note that the accuracy of approximating  $M_0$  is limited to  $\epsilon_d$  motivating the need for an “exact” distributed solver reducing the error to any  $\epsilon > 0$ .

### 3.1.2 “Exact” Distributed SDDM Solvers

Having introduced DistrRSolve, we are now ready to present a distributed version of Algorithm 2 which enables the computation of  $\epsilon$  close solutions for  $M_0 x = b_0$ . Contrary to the work of [11], our algorithm is capable of acquiring solutions up to any arbitrary  $\epsilon > 0$ . Similar to DistrRSolve, each node  $v_k \in \mathcal{V}$  receives the  $k^{th}$  row of  $M_0$ ,  $[b_0]_k$ ,  $d$  and a precision parameter  $\epsilon$  as inputs. Node  $v_k$  then computes the  $k^{th}$  component of the  $\epsilon$  close approximation of  $x^*$  by using DistrRSolve as a sub-routine and updates the solution iteratively as shown in lines 2-6 in Algorithm 4.

**Analysis of Algorithm 4:** Here, we again provide the theoretical analysis needed for quantifying the convergence and computational time of the exact algorithm for returning  $\epsilon$ -close approximation to  $x^*$ . The following lemma shows that DistrESolve computes the  $k^{th}$  component of the  $\epsilon$ -close approximation of  $x^*$ :

---

<sup>3</sup>For ease of presentation, we leave the proof of the lemma to the appendix.

---

**Algorithm 4** DistrESolve ( $\{[M_0]_{k1}, \dots, [M_0]_{kn}\}, [b_0]_k, d, \epsilon$ )

---

1: **Initialize:**  $[y_0]_k = 0$ ;  $[\chi]_k = \text{DistrRSolve}(\{[M_0]_{k1}, \dots, [M_0]_{kn}\}, [b_0]_k, d)$  (i.e., Algorithm 3)

2: **for**  $t = 1$  to  $q$  **do**

3:  $\begin{bmatrix} u_t^{(1)} \end{bmatrix}_k = [D_0]_{kk} [y_{t-1}]_k - \sum_{j: v_j \in \mathbb{N}_1(v_k)} [A_0]_{kj} [y_{t-1}]_j$

4:  $\begin{bmatrix} u_t^{(2)} \end{bmatrix}_k = \text{DistrRSolve}(\{[M_0]_{k1}, \dots, [M_0]_{kn}\}, \begin{bmatrix} u_t^{(1)} \end{bmatrix}_k, d,)$

5:  $[y_t]_k = [y_{t-1}]_k - \begin{bmatrix} u_t^{(2)} \end{bmatrix}_k + [\chi]_k$

6: **end for**

7:  $[\tilde{x}]_k = [y_q]_k$

8: **return**  $[\tilde{x}]_k$

---

**Lemma 8.** Let  $M_0 = D_0 - A_0$  be the standard splitting. Further, let  $\epsilon_d < \frac{1}{3} \ln 2$  in the nverse approximated chain  $\mathcal{C} = \{A_0, D_0, A_1, D_1, \dots, A_d, D_d\}$ . Then  $\text{DistrESolve}(\{[M_0]_{k1}, \dots, [M_0]_{kn}\}, [b_0]_k, d, \epsilon)$  requires  $\mathcal{O}(\log \frac{1}{\epsilon})$  iterations to return the  $k^{\text{th}}$  component of the  $\epsilon$  close approximation for  $x^*$ .

The above lemma proofs that the algorithm requires  $\mathcal{O}(\log \frac{1}{\epsilon})$  iterations for attaining for returning the  $k^{\text{th}}$  of the  $\epsilon$ -close approximation to  $x^*$ . Consequently, the overall complexity can be summarized as:

**Lemma 9.** Let  $M_0 = D_0 - A_0$  be the standard splitting. Further, let  $\epsilon_d < \frac{1}{3} \ln 2$  in the inverse approximated chain  $\mathcal{C} = \{A_0, D_0, A_1, D_1, \dots, A_d, D_d\}$ . Then,  $\text{DistrESolve}(\{[M_0]_{k1}, \dots, [M_0]_{kn}\}, [b_0]_k, d, \epsilon)$  requires  $\mathcal{O}(dn^2 \log(\frac{1}{\epsilon}))$  time steps.

### 3.1.3 Length of the Inverse Chain

Both introduced algorithms depend on the length of the inverse approximated chain,  $d$ . Here, we provide an analysis to determine the value of  $d$  which guarantees  $\epsilon_d < \frac{1}{3} \ln 2$  in  $\mathcal{C} = \{A_0, D_0, A_1, D_1, \dots, A_d, D_d\}$ :

**Lemma 10.** Let  $M_0 = D_0 - A_0$  be the standard splitting and  $\kappa$  denote the condition number of  $M_0$ . Consider the inverse approximated chain

$$\mathcal{C} = \{A_0, D_0, A_1, D_1, \dots, A_d, D_d\},$$

with  $d = \lceil \log \left( 2 \ln \left( \frac{\sqrt[3]{2}}{\sqrt[3]{2}-1} \right) \kappa \right) \rceil$ , then  $D_0 \approx_{\epsilon_d} D_0 - D_0 (D_0^{-1} A_0)^{2^d}$ , with  $\epsilon_d < \frac{1}{3} \ln 2$ .

*Proof.* The proof will be given as a collection of claims:

Claim: Let  $\kappa$  be the condition number of  $M_0 = D_0 - A_0$ , and  $\{\lambda_i\}_{i=1}^n$  denote the eigenvalues of  $D_0^{-1} A_0$ . Then,  $|\lambda_i| \leq 1 - \frac{1}{\kappa}$ , for all  $i = 1, \dots, n$

*Proof.* See Appendix. □

Notice that if  $\lambda_i$  represented an eigenvalue of  $D_0^{-1} A_0$ , then  $\lambda_i^r$  is an eigenvalue of  $(D_0^{-1} A_0)^r$  for all  $r \in \mathbb{N}$ . Therefore, we have

$$\rho \left( (D_0^{-1} A_0)^{2^d} \right) \leq \left( 1 - \frac{1}{\kappa} \right)^{2^d} \quad (9)$$

Claim: Let  $M$  be an SDDM matrix and consider the splitting  $M = D - A$ , with  $D$  being non negative diagonal and  $A$  being symmetric non negative. Further, assume that the eigenvalues of  $D^{-1} A$  lie between  $-\alpha$  and  $\beta$ . Then,

$$(1 - \beta)D \preceq D - A \preceq (1 + \alpha)D.$$

*Proof.* See Appendix. □

Combining the above results, gives

$$\left[1 - \left(1 - \frac{1}{\kappa}\right)^{2^d}\right] \mathbf{D}_d \preceq \mathbf{D}_d - \mathbf{A}_d \preceq \left[1 + \left(1 - \frac{1}{\kappa}\right)^{2^d}\right] \mathbf{D}_d.$$

Hence, to guarantee that  $\mathbf{D}_d \approx_{\epsilon_d} \mathbf{D}_d - \mathbf{A}_d$ , the following system must be satisfied:

$$e^{-\epsilon_d} \leq 1 - \left(1 - \frac{1}{\kappa}\right)^{2^d}, \quad \text{and} \quad e^{\epsilon_d} \geq 1 + \left(1 - \frac{1}{\kappa}\right)^{2^d}.$$

Introducing  $\gamma$  for  $\left(1 - \frac{1}{\kappa}\right)^{2^d}$ , we arrive at:

$$\epsilon_d \geq \ln\left(\frac{1}{1-\gamma}\right), \quad \text{and} \quad \epsilon_d \geq \ln(1+\gamma).$$

Hence,  $\epsilon_d \geq \max\left\{\ln\left(\frac{1}{1-\gamma}\right), \ln(1+\gamma)\right\} = \ln\left(\frac{1}{1-\gamma}\right)$ . Now, notice that if  $d = \lceil \log c\kappa \rceil$  then,  $\gamma = \left(1 - \frac{1}{\kappa}\right)^{2^d} = \left(1 - \frac{1}{\kappa}\right)^{c\kappa} \leq \frac{1}{e^c}$ . Hence,  $\ln\left(\frac{1}{1-\gamma}\right) \leq \ln\left(\frac{e^c}{e^c-1}\right)$ . This gives  $c = \lceil 2 \ln\left(\frac{\sqrt[3]{2}}{\sqrt[3]{2}-1}\right) \rceil$ , implying  $\epsilon_d = \ln\left(\frac{e^c}{e^c-1}\right) < \frac{1}{3} \ln 2$ . □

Using the above results the time complexity of DistrESolve with  $d = \lceil \log\left(2 \ln\left(\frac{\sqrt[3]{2}}{\sqrt[3]{2}-1}\right) \kappa\right) \rceil$  is  $\mathcal{O}\left(n^2 \log \kappa \log\left(\frac{1}{\epsilon}\right)\right)$  times steps, which concludes the proof of Theorem 1.

### 3.2 R-Hop Distributed Solvers

The above version of the distributed solver requires no knowledge of the graph's topology, but does require the information from all other nodes. Next, we will outline an R-Hop version of the algorithm in which communication is restricted to the R-Hop neighborhood between nodes. Due to such communication constraints, the R-Hop solver is general enough to be applied in a variety of fields including but not limited to, network flow problems (see Section 4). Along with Theorem 2, the following corollary summarizes the results of the R-Hop distributed solver:

**Corollary 1.** *Let  $\mathbf{M}_0$  be the weighted Laplacian of  $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathbf{W})$ . There exists a decentralized algorithm that uses only R-hop communication between nodes and computes  $\epsilon$  close solutions of  $\mathbf{M}_0 \mathbf{x} = \mathbf{b}_0$  in  $\mathcal{O}\left(\frac{n^3 \alpha}{R} \frac{\mathbf{W}_{\max}}{\mathbf{W}_{\min}} \log\left(\frac{1}{\epsilon}\right)\right)$  time steps, with  $n$  being the number of nodes in  $\mathcal{G}$ ,  $\mathbf{W}_{\max}$ ,  $\mathbf{W}_{\min}$*

*denoting the largest and the smallest weights of edges in  $\mathcal{G}$ , respectively,  $\alpha = \min\left\{n, \frac{(d_{\max}^{R+1}-1)}{(d_{\max}-1)}\right\}$*

*representing the upper bound on the size of the R-hop neighborhood  $\forall v \in \mathcal{V}$ , and  $\epsilon \in (0, \frac{1}{2}]$  being the precision parameter.*

Similar to development of the full communication solver, the R-Hop version also requires two steps to attain the  $\epsilon$ -close approximation to  $\mathbf{x}^*$ , i.e., the “crude R-Hop” and the “exact R-Hop” solutions.

#### 3.2.1 “Crude” R-Hop Distributed Solver

The “crude R-Hop” solver uses the same inverse approximated chain as that of the full communication version (see Equation 8) to acquire a “crude” approximation for the  $k^{th}$  component  $\mathbf{x}_0$  while only requiring R-Hop communication between the nodes. Algorithm 5 represents the “crude” R-Hop solver requiring the inverse chain,  $k^{th}$  component of  $\mathbf{b}_0$ , length of the inverse chain  $d$ , and the communication bound  $R$  as inputs. Namely, each node  $v_k \in \mathcal{V}$  receives the  $k^{th}$  row of  $\mathbf{M}_0$ ,  $k^{th}$  component,  $[\mathbf{b}_0]_k$ , of  $\mathbf{b}_0$ , the length of the inverse chain,  $d$ , and the local communication bound<sup>4</sup>  $R$  as inputs to output the  $k^{th}$  component of the “crude” approximation to  $\mathbf{x}^*$ . Algorithm 5 operates in three major parts. Due to the need of the R-powers of  $\mathbf{A}_0 \mathbf{D}_0^{-1}$  and  $\mathbf{D}_0 \mathbf{A}_0^{-1}$ , the first step is to compute such matrices in a distributed manner.

<sup>4</sup>For simplicity,  $R$  is assumed to be in the order of powers of 2, i.e.,  $R = 2^\rho$ .

---

**Algorithm 5** RDistRSolve ( $\mathcal{C}, [\mathbf{b}_0]_k, d, R$ )

---

```
1: Part One:
2:  $\{[C_0]_{k1}, \dots, [C_0]_{kn}\} = f_0([M_0]_{k1}, \dots, [M_0]_{kn}, R)$ 
3:  $\{[C_1]_{k1}, \dots, [C_1]_{kn}\} = f_1([M_0]_{k1}, \dots, [M_0]_{kn}, R)$ 

4: Part Two:
5: for  $i = 1$  to  $d$  do
6:    $l_{i-1} = 2^{i-1}/R$ 
7:    $[u_1^{(i-1)}]_k = [A_0 D_0^{-1} \mathbf{b}_{i-1}]_k$ 
8:    $[u_{l_{i-1}}^{(i-1)}]_k = f_2([u_1^{(i-1)}]_k)$ 
9:    $[b_i]_k = [b_{i-1}]_k + [u_{l_{i-1}}^{(i-1)}]_k$ 
10: end for

11: Part Three:
12:  $[x_d]_k = [b_d]_k / [D_0]_{kk}$ 
13: for  $i = d - 1$  to  $1$  do
14:    $l_i = 2^i/R$ 
15:    $[\eta_1^{(i+1)}]_k = [D_0^{-1} A_0 x_{i+1}]_k$ 
16:    $[\eta_{l_i}^{(i+1)}]_k = f_3([\eta_1^{(i+1)}]_k)$ 
17:    $[x_i]_k = \frac{1}{2} \left[ \frac{[b_i]_k}{[D_0]_{kk}} + [x_{i+1}]_k + [\eta_{l_i}^{(i+1)}]_k \right]$ 
18: end for
19:  $[x_0]_k = \frac{1}{2} \left[ \frac{[b_0]_k}{[D_0]_{kk}} + [x_1]_k + [D_0^{-1} A_0 x_1]_k \right]$ 
20: return  $[x_0]_k$ 
```

---

---

**Algorithm 6**  $f_0([M_0]_{k1}, \dots, [M_0]_{kn}, R)$ 

---

```
1: for  $l = 1$  to  $R - 1$  do
2:   for  $j$  s.t.  $\mathbf{v}_j \in \mathbb{N}_{l+1}(\mathbf{v}_k)$  do
3:      $[(A_0 D_0^{-1})^{l+1}]_{kj} = \sum_{r: \mathbf{v}_r \in \mathbb{N}_1(\mathbf{v}_j)} \frac{[D_0]_{rr}}{[D_0]_{jj}} [(A_0 D_0^{-1})^l]_{kr} [A_0 D_0^{-1}]_{jr}$ 
4:   end for
5: end for
6: return  $c_0 = \{[(A_0 D_0^{-1})^R]_{k1}, \dots, [(A_0 D_0^{-1})^R]_{kn}\}$ 
```

---

Given the inverse chain and the communication bound,  $R$ ,  $f_0(\cdot)$  and  $f_1(\cdot)$  serve this cause as detailed in Algorithms 6 and 7, respectively. Essentially, these algorithms execute multiplications needed for determining  $(A D_0^{-1})^R$  and  $(D A_0^{-1})^R$  in a distributed fashion looping over the relevant hops of the network. For a node,  $\mathbf{v}_k \in \mathcal{V}$ , the  $k^{th}$  component of these powers are returned to Algorithm 5 as  $[C_0]_{ki} = [(A_0 D_0^{-1})^R]_{ki}$  and  $[C_1]_{ki} = [(D_0 A_0^{-1})^R]_{ki}$  for  $i = \{1, \dots, n\}$ ; see Part One in Algorithm 5.

Similar to the full communication version, the second two parts of the “crude R-Hop” solver run two loops. In the first, the  $k^{th}$  component of  $\mathbf{b}_i$  is computed by looping forward through the inverse chain, while in the second the  $k^{th}$  component of the crude solution is determined by looping backwards.

The second part of the solver is better depicted in the flow diagrams of Figures 1(a) and 1(b). Within the first loop running through the length of the inverse chain, the condition  $i - 1 < \rho$  is checked. In case this condition is true,  $A_0$ ,  $D_0$ , and the previous iteration vector  $\mathbf{b}_{i-1}$  are used to update  $\mathbf{b}_i$  as shown in Figure 1(a).

$$[b_i]_k = [b_{i-1}]_k + [u_{l_{i-1}}^{(i-1)}]_k$$

---

**Algorithm 7**  $f_1([M_0]_{k1}, \dots, [M_0]_{kn}, R)$ 


---

```

1: for  $l = 1$  to  $R - 1$  do
2:   for  $j$  s.t.  $v_j \in \mathbb{N}_{l+1}(v_k)$  do
3:      $[(D_0^{-1}A_0)^{l+1}]_{kj} = \sum_{r: v_r \in \mathbb{N}_1(v_j)} \frac{[D_0]_{jj}}{[D_0]_{rr}} [(D_0^{-1}A_0)^l]_{kr} [D_0^{-1}A_0]_{jr}$ 
4:   end for
5: end for
6: return  $c_1 = \{[(D_0^{-1}A_0)^R]_{k1}, \dots, [(D_0^{-1}A_0)^R]_{kn}\}$ 

```

---

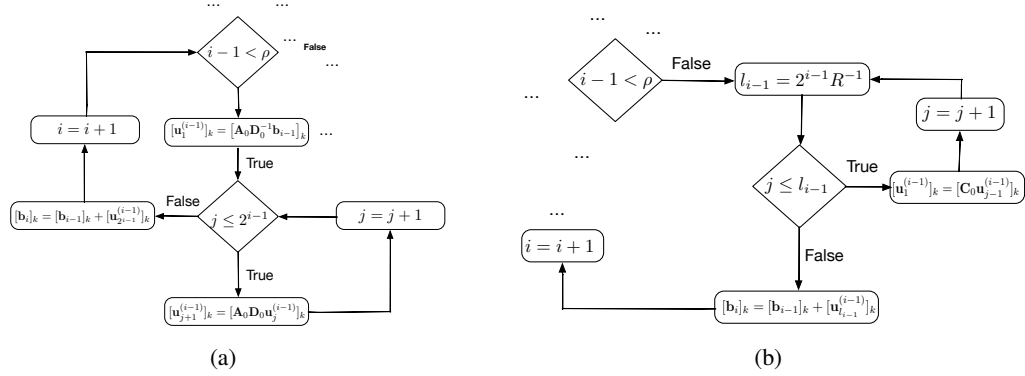


Figure 1: The left figure depicts the first condition of part two in Algorithm 5 which has to be checked. In case  $i - 1 < \rho$ , the computations shown in the figure are executed. The right figure handles the case when  $i - 1 \geq \rho$ . Here, the computations shown in the figure are executed. Note, that the chains are constructed using the computations returned by  $f_0(\cdot)$  and  $f_1(\cdot)$ .

This is performed using another loop constructing a series of  $[u_j^{(i-1)}]_k$  vectors for  $j = 1, \dots, 2^{i-1}$ , used to update the  $k^{th}$  component of  $b_i$  at the  $i^{th}$  iteration: At the next  $i^{th}$  iteration, the condition  $i - 1 < \rho$  is checked again. In case this condition is met, the previous computations are executed again. Otherwise, the commands depicted in Figure 1(b) run. Here, a temporary variable denoting the fraction to the communication bound  $R$ ,  $l_{i-1} = \frac{2^{i-1}}{R}$ , is used to determine the upper iterate bound in  $j$ . Again throughout this loop, a series of  $u$  vectors used to update  $[b_i]_k$  are constructed.

Having terminated the forward loop, the  $k^{th}$  component of the “crude” solution is computed by looping backward through the inverse approximated chain (see part three in Algorithm 5). For  $i$  running backwards to 1, an R-Hop condition,  $\rho$ , is checked. In case  $i < \rho$ , the following update is performed:

$$[\eta_j^{(i+1)}]_k = [D_0^{-1}A_0\eta_{j-1}^{(i+1)}]_k,$$

for  $j = 2, \dots, 2^i$ ,  $i = d - 1, \dots, 1$ , and

$$[\eta_1^{(i+1)}]_k = [D_0^{-1}A_0x_{i+1}]_k.$$

The role of  $\eta$  are backward intermediate solutions needed for updating the “crude” solution to  $M_0x = b_0$ :

$$[x_i]_k = \frac{1}{2} \left[ \frac{[b_i]_k}{[D_0]_{kk}} + [x_1]_k + [\eta_{2^i}^{(i+1)}]_k \right],$$

for a node  $v_k \in \mathcal{V}$ . In case  $i \geq \rho$  a similar set of computations are executed for updating the crude solution using:

$$[x_i]_k = \frac{1}{2} \left[ \frac{[b_i]_k}{[D_0]_{kk}} + [x_1]_k + [D_0^{-1}Ax_1]_k \right].$$

**Analysis of Algorithm 5** Similar to the previous section, we next provide the theoretical analysis needed for quantifying the performance of the crude R-Hop solver. The following Lemma shows

that  $\text{RDistRSolve}$  computes the  $k^{th}$  component of the “crude” approximation of  $\mathbf{x}^*$  and provides the algorithm’s time complexity:

**Lemma 11.** *Let  $\mathbf{M}_0 = \mathbf{D}_0 - \mathbf{A}_0$  be the standard splitting and let  $\mathbf{Z}'_0$  be the operator defined by  $\text{RDistRSolve}$ , namely,  $\mathbf{x}_0 = \mathbf{Z}'_0 \mathbf{b}_0$ , then  $\mathbf{Z}'_0 \approx_{\epsilon_d} \mathbf{M}_0^{-1}$ . Furthermore,  $\text{RDistRSolve}$  requires*

$$\mathcal{O} \left( \frac{2^d}{R} \alpha + \alpha R d_{max} \right),$$

where  $\alpha = \min \left\{ n, \frac{(d_{max}^{R+1} - 1)}{(d_{max} - 1)} \right\}$  to arrive at  $\mathbf{x}_0$ .

*Proof.* The proof of the above Lemma can be obtained by proving a collection of claims:

Claim: Matrices  $(\mathbf{D}_0^{-1} \mathbf{A}_0)^r$  and  $(\mathbf{A}_0 \mathbf{D}_0^{-1})^r$  have sparsity patterns corresponding to the  $r$ -Hop neighborhood for any  $r \in \mathbb{N}$ .

The above claim is proved by induction on  $R$ . We start with the base case: for  $R = 1$ ,

$$[\mathbf{A}_0 \mathbf{D}_0^{-1}]_{ij} = \begin{cases} \frac{[\mathbf{A}_0]_{ij}}{[\mathbf{D}_0]_{ii}} & \text{if } j : \mathbf{v}_j \in \mathbb{N}_1(\mathbf{v}_i) \\ 0 & \text{otherwise.} \end{cases}$$

Therefore,  $\mathbf{A}_0 \mathbf{D}_0^{-1}$  has a sparsity pattern corresponding to the 1-Hop neighborhood. Assume that for all  $1 \leq p \leq R-1$ ,  $(\mathbf{A}_0 \mathbf{D}_0^{-1})^p$  has a sparsity pattern corresponding to the  $p$ -hop neighborhood. Consider,  $(\mathbf{A}_0 \mathbf{D}_0^{-1})^R$

$$[(\mathbf{A}_0 \mathbf{D}_0^{-1})^R]_{ij} = \sum_{k=1}^n [(\mathbf{A}_0 \mathbf{D}_0^{-1})^{R-1}]_{ik} [\mathbf{A}_0 \mathbf{D}_0^{-1}]_{kj} \quad (10)$$

Since  $\mathbf{A}_0 \mathbf{D}_0^{-1}$  is non negative, then  $[(\mathbf{A}_0 \mathbf{D}_0^{-1})^R]_{ij} \neq 0$  iff there exists  $k$  such that  $\mathbf{v}_k \in \mathbb{N}_{R-1}(\mathbf{v}_i)$  and  $\mathbf{v}_k \in \mathbb{N}_1(\mathbf{v}_j)$ , namely,  $\mathbf{v}_j \in \mathbb{N}_R(\mathbf{v}_i)$ . The proof can be done in a similar fashion for  $\mathbf{D}_0^{-1} \mathbf{A}_0$ .  $\square$

The next claim provides complexity guarantees for  $f_0(\cdot)$  and  $f_1(\cdot)$  described in Algorithms 6 and 7, respectively.

Claim: Algorithms 6 and 7 use only the  $R$ -hop information to compute the  $k^{th}$  row of  $(\mathbf{D}_0^{-1} \mathbf{A}_0)^R$  and  $(\mathbf{A}_0 \mathbf{D}_0^{-1})^R$ , respectively, in  $\mathcal{O}(\alpha R d_{max})$  time steps, where  $\alpha = \min \left\{ n, \frac{(d_{max}^{R+1} - 1)}{(d_{max} - 1)} \right\}$ .

*Proof.* The proof will be given for  $f_0(\cdot)$  described in Algorithm 6 as that for  $f_1(\cdot)$  can be performed similarly. Due to Claim 3.2.1, we have

$$\left[ (\mathbf{A}_0 \mathbf{D}_0^{-1})^{l+1} \right]_{kj} = \sum_{r=1}^n \left[ (\mathbf{A}_0 \mathbf{D}_0^{-1})^l \right]_{kr} [\mathbf{A}_0 \mathbf{D}_0^{-1}]_{rj} = \sum_{r: \mathbf{v}_r \in \mathbb{N}_1(\mathbf{v}_j)} \left[ (\mathbf{A}_0 \mathbf{D}_0^{-1})^l \right]_{kr} [\mathbf{A}_0 \mathbf{D}_0^{-1}]_{rj} \quad (11)$$

Therefore at iteration  $l+1$ ,  $\mathbf{v}_k$  computes the  $k^{th}$  row of  $(\mathbf{A}_0 \mathbf{D}_0^{-1})^{l+1}$  using:

- (1) the  $k^{th}$  row of  $(\mathbf{A}_0 \mathbf{D}_0^{-1})^l$ , and
- (2) the  $r^{th}$  column of  $\mathbf{A}_0 \mathbf{D}_0^{-1}$ .

Node  $\mathbf{v}_r$ , however, can only send the  $r^{th}$  row of  $\mathbf{A}_0 \mathbf{D}_0^{-1}$  making  $\mathbf{A}_0 \mathbf{D}_0^{-1}$  non-symmetric. Noting that  $[\mathbf{A}_0 \mathbf{D}_0^{-1}]_{rj} / [\mathbf{D}_0]_{rr} = [\mathbf{A}_0 \mathbf{D}_0^{-1}]_{jr} / [\mathbf{D}_0]_{jj}$ , since  $\mathbf{D}_0^{-1} \mathbf{A}_0 \mathbf{D}_0^{-1}$  is symmetric, leads to  $[(\mathbf{A}_0 \mathbf{D}_0^{-1})^{l+1}]_{kj} = \sum_{r: \mathbf{v}_r \in \mathbb{N}_1(\mathbf{v}_j)} \frac{[\mathbf{D}_0]_{rr}}{[\mathbf{D}_0]_{jj}} [(\mathbf{A}_0 \mathbf{D}_0^{-1})^l]_{kr} [\mathbf{A}_0 \mathbf{D}_0^{-1}]_{jr}$ . To prove the time complexity

guarantee, at each iteration  $\mathbf{v}_k$  computes at most  $\alpha$  values, where  $\alpha = \min \left\{ n, \frac{(d_{max}^{R+1} - 1)}{(d_{max} - 1)} \right\}$  is the upper bound on the size of the  $R$ -hop neighborhood  $\forall \mathbf{v} \in \mathcal{V}$ . Each such computation requires at most  $\mathcal{O}(d_{max})$  operations. Thus, the overall time complexity is given by  $\mathcal{O}(\alpha R d_{max})$ .  $\square$

We are now ready to provide the proof of Lemma 11.

*Proof.* From **Parts Two** and **Three** of Algorithm 5, it is clear that node  $v_k$  computes  $[b_1]_k, [b_2]_k, \dots, [b_d]_k$  and  $[x_d]_k, [x_{d-1}]_k, \dots, [x_0]_k$ , respectively. These are determined using the inverse approximated chain as follows

$$\begin{aligned} b_i &= (I + (A_{i-1}D_{i-1}^{-1})b_{i-1} = b_{i-1} + (A_0D_0^{-1})^{2^{i-1}}b_{i-1} \\ x_i &= \frac{1}{2}[D_i^{-1}b_i + (I + D_i^{-1}A_i)x_{i+1}] = \frac{1}{2}[D_0^{-1}b_i + x_{i+1} + (D_0^{-1}A_0)^{2^i}x_{i+1}] \end{aligned} \quad (12)$$

Considering the computation of  $[b_1]_k, \dots, [b_d]_k$  for  $\rho > i - 1$ , we have

$$\begin{aligned} [b_i]_k &= [b_{i-1}]_k + [(A_0D_0^{-1})^{2^{i-1}}b_{i-1}]_k = [b_{i-1}]_k + \underbrace{[A_0D_0^{-1} \dots A_0D_0^{-1}b_{i-1}]_k}_{2^{i-1}} \\ &= [b_{i-1}]_k + \underbrace{[A_0D_0^{-1} \dots A_0D_0^{-1}u_1^{(i-1)}]_k}_{2^{i-1}-1} \dots = [b_{i-1}]_k + \left[ u_{2^{i-1}}^{(i-1)} \right]_k \end{aligned}$$

with  $u_{j+1}^{(i-1)} = A_0D_0^{-1}u_j^{(i-1)}$  for  $j = 1, \dots, 2^{i-1} - 1$ . Since  $A_0D_0^{-1}$  has a sparsity pattern corresponding to 1-hop neighborhood (see Claim 3.2.1), node  $v_k$  computes  $\left[ u_{j+1}^{(i-1)} \right]_k$ , based on  $u_j^{(i-1)}$ , acquired from its 1-Hop neighbors. It is easy to see that  $\forall i$  such that  $i - 1 < \rho$  the computation of  $[b_i]_k$  requires  $\mathcal{O}(2^{i-1}d_{max})$  time steps. Thus, the computation of  $[b_1]_k, \dots, [b_\rho]_k$  requires  $\mathcal{O}(2^\rho d_{max}) = \mathcal{O}(Rd_{max})$ . Now, consider the computation of  $[b_i]_k$  but for  $i - 1 \geq \rho$

$$\begin{aligned} [b_i]_k &= [b_{i-1}]_k + [(A_0D_0^{-1})^{2^{i-1}}b_{i-1}]_k = [b_{i-1}]_k + \underbrace{[C_0 \dots C_0b_{i-1}]_k}_{l_{i-1}} \\ &= [b_{i-1}]_k + \underbrace{[C_0 \dots C_0u_1^{(i-1)}]_k}_{l_{i-1}-1} = [b_{i-1}]_k + \left[ u_{l_{i-1}}^{(i-1)} \right]_k \end{aligned}$$

with  $C_0 = (A_0D_0^{-1})^R$ ,  $l_{i-1} = \frac{2^{i-1}}{R}$ , and  $u_{j+1}^{(i-1)} = C_0u_j^{(i-1)}$  for  $j = 1, \dots, l_{i-1} - 1$ . Since  $C_0$  has a sparsity pattern corresponding to R-hop neighborhood (see Claim 3.2.1), node  $v_k$  computes  $\left[ u_{j+1}^{(i-1)} \right]_k$  based on the components of  $u_j^{(i-1)}$  attained from its R-hop neighbors. For each  $i$  such that  $i - 1 \geq \rho$  the computing  $[b_i]_k$  requires  $\mathcal{O}\left(\frac{2^{i-1}}{R}\alpha\right)$  time steps, where  $\alpha = \min\left\{n, \frac{(d_{max}^{R+1}-1)}{(d_{max}-1)}\right\}$  being the upper bound on the number of nodes in the  $R$ -hop neighborhood  $\forall v \in \mathcal{V}$ . Therefore, the overall computation of  $[b_{\rho+1}]_k, [b_{\rho+2}]_k, \dots, [b_d]_k$  is achieved in  $\mathcal{O}\left(\frac{2^d}{R}\alpha\right)$  time steps. Finally, the time complexity for the computation of all of the values  $[b_1]_k, [b_2]_k, \dots, [b_d]_k$  is  $\mathcal{O}\left(\frac{2^d}{R}\alpha + Rd_{max}\right)$ . Similar analysis can be applied to determine the computational complexity of  $[x_d]_k, [x_{d-1}]_k, \dots, [x_1]_k$ , i.e., **Part Three** of Algorithm 5. We arrive at  $Z'_0 \approx_{\epsilon_d} M_0^{-1}$ . Finally, using Claim 3.2.1, the time complexity of RDistRSolve (Algorithm 5) is  $\mathcal{O}\left(\frac{2^d}{R}\alpha + \alpha Rd_{max}\right)$ .  $\square$

### 3.2.2 “Exact” R-Hop Distributed Solver

Having developed an R-hop version which computes a “crude” approximation to the solution of  $M_0x = b_0$ , now we provide an exact R-hop solver presented in Algorithm 8. Similar to RDistRSolve, each node  $v_k$  receives the  $k^{th}$  row  $M_0$ ,  $[b_0]_k$ ,  $d$ ,  $R$ , and a precision parameter  $\epsilon$  as inputs, and outputs the  $k^{th}$  component of the  $\epsilon$  close approximation of vector  $x^*$ .

**Analysis of Algorithm 8:** The following Lemma shows that EDistRSolve computes the  $k^{th}$  component of the  $\epsilon$  close approximation to  $x^*$  and provides the time complexity analysis.

**Lemma 12.** *Let  $M_0 = D_0 - A_0$  be the standard splitting. Further, let  $\epsilon_d < 1/3 \ln 2$ . Then Algorithm 8 requires  $\mathcal{O}\left(\log \frac{1}{\epsilon}\right)$  iterations to return the  $k^{th}$  component of the  $\epsilon$  close approximation to  $x^*$ .*

---

**Algorithm 8** EDistRSolve ( $\{[M_0]_{k1}, \dots, [M_0]_{kn}\}, [b_0]_k, d, R, \epsilon$ )

---

**Initialize:**  $[y_0]_k = 0$ , and  $[\chi]_k = \text{RDistRSolve}(\{[M_0]_{k1}, \dots, [M_0]_{kn}\}, [b_0]_k, d, R)$   
**for**  $t = 1$  to  $q$  **do**  
 $[u_t^{(1)}]_k = [D_0]_{kk}[y_{t-1}]_k - \sum_{j: v_j \in \mathbb{N}_1(v_k)} [A_0]_{kj}[y_{t-1}]_j$   
 $[u_t^{(2)}]_k = \text{RDistRSolve}(\{[M_0]_{k1}, \dots, [M_0]_{kn}\}, [u_t^{(1)}]_k, d, R)$   
 $[y_t]_k = [y_{t-1}]_k - [u_t^{(2)}]_k + [\chi]_k$   
**end for**  
**return**  $[\tilde{x}]_k = [y_q]_k$

---

The following Lemma provides the time complexity analysis of EDistRSolve:

**Lemma 13.** *Let  $M_0 = D_0 - A_0$  be the standard splitting and let  $\epsilon_d < 1/3 \ln 2$ , then EDistRSolve requires  $\mathcal{O}((2^d/R\alpha + \alpha R d_{\max}) \log(1/\epsilon))$  time steps. Moreover, for each node  $v_k$ , EDistRSolve only uses information from the  $R$ -hop neighbors.*

**Length of the Inverse Chain:** Again these introduced algorithms depend on the length of the inverse approximated chain,  $d$ . The analysis in Section 3.1.3 can be applied again to determine the  $d = \lceil \log \left( 2 \ln \left( \frac{\sqrt[3]{2}}{\sqrt[3]{2}-1} \right) \kappa \right) \rceil$  as the length of the inverse chain.

### 3.3 Comparison to Existing Literature

As mentioned before, the proposed solver is a distributed version of the parallel SDDM solver of [11]. Our approach is capable of acquiring  $\epsilon$ -close solutions for arbitrary  $\epsilon > 0$  in  $\mathcal{O}\left(n^3 \frac{\alpha}{R} \frac{W_{\max}}{W_{\min}} \log\left(\frac{1}{\epsilon}\right)\right)$ , with  $n$  the number of nodes in graph  $\mathcal{G}$ ,  $W_{\max}$  and  $W_{\min}$  denoting the largest and smaller weights of the edges in  $\mathcal{G}$ , respectively,  $\alpha = \min\left\{n, \frac{d_{\max}^{R+1}-1}{d_{\max}-1}\right\}$  representing the upper bound on the size of the  $R$ -Hop neighborhood  $\forall v \in \mathcal{V}$ , and  $\epsilon \in (0, \frac{1}{2}]$  as the precision parameter. After developing the full communication version, we proposed a generalization to the  $R$ -Hop case where communication is restricted.

Our method is faster than state-of-the-art methods for iteratively solving linear systems. Typical linear methods, such as Jacobi iteration [16], are guaranteed to converge if the matrix is *strictly* diagonally dominant. We proposed a distributed algorithm that generalizes this setting, where it is guaranteed to converge in the SDD/SDDM scenario. Furthermore, the time complexity of linear techniques is  $\mathcal{O}(n^{1+\beta} \log n)$ , hence, a case of strictly diagonally dominant matrix  $M_0$  can be easily constructed to lead to a complexity of  $\mathcal{O}(n^4 \log n)$ . Consequently, our approach not only generalizes the assumptions made by linear methods, but is also faster by a factor of  $\log n$ . Furthermore, such algorithms require average consensus to decentralize vector norm computations. Contrary to these methods which lead to additional approximation errors to the real solution, our approach resolves these issues by eliminating the need for such a consensus framework.

In centralized solvers, nonlinear methods (e.g., conjugate gradient descent [37, 36], etc.) typically offer computational advantages over linear methods (e.g., Jacobi Iteration) for iteratively solving linear systems. These techniques, however, can not be easily decentralized. For instance, the stopping criteria for nonlinear methods require the computation of weighted norms of residuals (e.g.,  $\|p_k\|_{M_0}$  with  $p_k$  being the search direction at iteration  $k$ ). To the best of our knowledge, the distributed computation of weighted norms is difficult. Namely using the approach in [38], this requires the calculation of the top singular value of  $M_0$  which amounts to a power iteration on  $M_0^\top M_0$  leading to the loss of sparsity. Furthermore, conjugate gradient methods require global computations of inner products.

Another existing method which we compare our results to is the recent work of the authors [34] where a local and asynchronous solution for solving systems of linear equations is considered. In their work, the authors derive a complexity bound, for one component of the solution vector, of  $\mathcal{O}\left(\min\left(d\epsilon^{\frac{1}{\ln d}} \frac{1}{\|\mathcal{G}\|_2}, \frac{dn \ln \epsilon}{\ln \|\mathcal{G}\|_2}\right)\right)$ , with  $\epsilon$  being the precision parameter,  $d$  a constant bound on the maximal degree of  $\mathcal{G}$ , and  $\mathcal{G}$  is defined as  $x = \mathcal{G}x + z$  which can be directly mapped to  $Ax = b$ . The relevant scenario to our work is when  $A$  is PSD and  $\mathcal{G}$  is symmetric. Here, the bound on

the number of multiplications is given by  $\mathcal{O}\left(\min\left(d^{\frac{\kappa(\mathbf{A})+1}{2}} \ln \frac{1}{\epsilon}, \frac{\kappa(\mathbf{A})+1}{2} nd \ln \frac{1}{\epsilon}\right)\right)$ , with  $\kappa(\mathbf{A})$  being the condition number of  $\mathbf{A}$ . In the general case, when the degree depends on the number of nodes (i.e.,  $d = d(n)$ ), the minimum in the above bound will be the result of the second term ( $\frac{\kappa(\mathbf{A})+1}{2} nd \ln \frac{1}{\epsilon}$ ) leading to  $\mathcal{O}\left(d(n)n\kappa(\mathbf{A}) \ln \frac{1}{\epsilon}\right)$ . Consequently, in such a general setting, our approach outperforms [34] by a factor of  $d(n)$ .

**Special Cases:** To better understand the complexity of the proposed SDDM solvers, next we detail the complexity for three specific graph structures. Before deriving these special cases, however, we first note the following simple yet useful connection between weighted and unweighted Laplacians of a graph  $\mathcal{G}$ . Denoting by  $\mathbf{B}$  the incidence matrix of  $\mathcal{G}$  and  $\mathbf{W}$  a diagonal matrix with edge weights as diagonal elements, we can write:

$$\mathcal{L}_{\mathcal{G}} = \mathbf{B}^T \mathbf{B} \quad \text{and} \quad \mathcal{L}_{\mathcal{G}}^{(\text{weighted})} = \mathbf{B}^T \mathbf{W} \mathbf{B}.$$

Hence, we can easily establish:

$$\mu_n\left(\mathcal{L}_{\mathcal{G}}^{(\text{weighted})}\right) \leq w_{\max} \mu_n\left(\mathcal{L}_{\mathcal{G}}\right) \quad \text{and} \quad \mu_2\left(\mathcal{L}_{\mathcal{G}}^{(\text{weighted})}\right) \geq w_{\min}\left(\mathcal{L}_{\mathcal{G}}\right).$$

This implies that the condition number of the weighted Laplacian satisfies:

$$\kappa\left(\mathcal{L}_{\mathcal{G}}^{(\text{weighted})}\right) = \frac{\mu_n\left(\mathcal{L}_{\mathcal{G}}^{(\text{weighted})}\right)}{\mu_2\left(\mathcal{L}_{\mathcal{G}}^{(\text{weighted})}\right)} \leq \frac{w_{\max}}{w_{\min}} \kappa\left(\mathcal{L}_{\mathcal{G}}\right),$$

with  $w_{\min}$  and  $w_{\max}$  are the minimal and maximal edge weights of  $\mathcal{G}$ . Using the above, we now consider four different graph topologies:

### 3.3.1 Path Graph

Similar to the hitting time of a Markov chain on a path graph which is given by  $\mathcal{O}(n^2)$ , the time complexity of the R-Hop SDDM solver is given by<sup>5</sup>:

**Corollary 2.** *Given a path graph  $\mathcal{P}_n$  with  $n$  nodes, the time complexity of the R-Hop SDDM solver is given by:*

$$T_{SDDM}(\mathcal{P}_n) = \mathcal{O}\left(n^2 \log \frac{1}{\epsilon}\right),$$

for any  $\epsilon > 0$  and for  $k = m = \mathcal{O}(\sqrt{n})$ .

### 3.3.2 Grid Graph

Recognizing that a grid graph  $\mathcal{G}_{k \times m}$  can be represented as a product of two path graphs,  $\mathcal{G}_{k \times m} = \mathcal{P}_k \times \mathcal{P}_m$ , our solver's computational time can be summarized by:

**Corollary 3.** *Given a grid graph,  $\mathcal{G}_{k \times m}$ , the time complexity of the distributed SDDM-solver can be bounded by:*

$$T_{SDDM}(\mathcal{G}_{k \times m}) = \mathcal{O}\left(n \log \frac{1}{\epsilon}\right),$$

for any  $\epsilon > 0$ .

### 3.3.3 Scale-Free Networks (Polya-Urn Graphs)

Using the results developed in [39, 40] the total time complexity of the distributed R-Hop solver is bounded by:

**Corollary 4.** *Given a scale-free network,  $\mathcal{G}_{SN(n)}$ , the time complexity of the R-Hop SDD solver for  $R=1$  is given by:*

$$T_{SDD}(\mathcal{L}_{\mathcal{G}_{SN(n)}}) = \mathcal{O}\left(n^2 \log n \log \frac{1}{\epsilon}\right).$$

---

<sup>5</sup>Due to space constraints, the proofs can be found in the appendix.

### 3.3.4 $d$ - Regular Ramanujan Expanders

For  $d$  regular Ramanujan expanders in which  $d$  does not depend on  $n$ , we have  $\mu_n(\mathcal{L}_{\text{RExp}(d)}) \leq 2d$  and  $\mu_2(\mathcal{L}_{\text{RExp}(d)}) \geq d - 2\sqrt{d-1}$ . Hence, the time complexity of the SDD-solver is given by constant time.

The developed R-Hop distributed SDDM solver is a fundamental contribution with wide ranging applicability. Next, we develop one such application from. We apply our solver for proposing an efficient and accurate distributed Newton method for network flow optimization. Namely, the distributed SDDM solver is used for computing the Newton direction in a distributed fashion up-to any arbitrary  $\epsilon > 0$ . This results in a novel distributed Newton method outperforming state-of-the-art techniques in both computational complexity and accuracy.

## 4 Distributed Newton Method for Network Flow Optimization

Conventional methods for distributed network optimization are based on sub-gradient descent in either the primal or dual domains, see. For a large class of problems, these techniques yield iterations that can be implemented in a distributed fashion using only local information. Their applicability, however, is limited by increasingly slow convergence rates. Second order Newton methods [4] are known to overcome this limitation leading to improved convergence rates.

Unfortunately, computing exact Newton directions based only on local information is challenging. Specifically, to determine the Newton direction, the inverse of the dual Hessian is needed. Determining this inverse, however, requires global information. Consequently, authors in [5, 6, 7] proposed approximate algorithms for determining these Newton iterates in a distributed fashion. Accelerated Dual Descent (ADD) [5], for instance, exploits the fact that the dual Hessian is the weighted Laplacian of the network and performs a truncated Neumann expansion of the inverse to determine a local approximate to the exact direction. ADD allows for a tradeoff between accurate Hessian approximations and communication costs through the N-Hop design, where increased N allows for more accurate inverse approximations arriving at increased cost, and lower values of N reduce accuracy but improve computational times. Though successful, the effectiveness of these approaches highly depend on the accuracy of the truncated Hessian inverse which is used to approximate the Newton direction. As shown later, the approximated iterate can resemble high variation to the real Newton direction, decreasing the applicability of these techniques.

**Contributions:** Exploiting the sparsity pattern of the dual Hessian, here we tackle the above problem and propose a Newton method for network optimization that is both faster and more accurate. Using the above developed solvers for SDDM linear equations, we approximate the Newton direction up-to any arbitrary precision  $\epsilon > 0$ . This leads to a distributed second-order method which performs almost identically the exact Newton method. Contrary to current distributed Newton methods, our algorithm is the first which is capable of attaining an  $\epsilon$ -close approximation to the Newton direction up to any arbitrary  $\epsilon > 0$ . We analyze the properties of the proposed algorithm and show that, similar to conventional Newton methods, superlinear convergence within a neighborhood of the optimal value is attained.

We finally demonstrate the effectiveness of the approach in a set of experiments on randomly generated and Barbell networks. Namely, we show that our method is capable of significantly outperforming state-of-the-art methods in both the convergence speeds and in the accuracy of approximating the Newton direction.

### 4.1 Network Flow Optimization

We consider a network represented by a directed graph  $\mathcal{G} = (\mathcal{N}, \mathcal{E})$  with node set  $\mathcal{N} = \{1, \dots, N\}$  and edge set  $\mathcal{E} = \{1, \dots, E\}$ . The flow vector is denoted by  $\mathbf{x} = [x^{(e)}]_{e \in \mathcal{E}}$ , with  $x^{(e)}$  representing the flow on edge  $e$ . The flow conservation conditions at nodes can be compactly represented as

$$\mathbf{A}\mathbf{x} = \mathbf{b},$$

where  $\mathbf{A}$  is the  $N \times E$  node-edge incidence matrix of  $\mathcal{G}$  defined as

$$A_{i,j} = \begin{cases} 1 & \text{if edge } j \text{ leaves node } i \\ -1 & \text{if edge } j \text{ enters node } i \\ 0 & \text{otherwise,} \end{cases}$$

and the vector  $\mathbf{b} \in \mathbf{1}^\perp$  denotes the external source, i.e.,  $b^{(i)} > 0$  (or  $b^{(i)} < 0$ ) indicates  $b^{(i)}$  units of external flow enters (or leaves) node  $i$ . A cost function  $\Phi_e : \mathbb{R} \rightarrow \mathbb{R}$  is associated with each edge  $e$ . Namely,  $\Phi_e(x^{(e)})$  denotes the cost on edge  $e$  as a function of the edge flow  $x^{(e)}$ . We assume that the cost functions  $\Phi_e$  are strictly convex and twice differentiable. Consequently, the minimum cost network optimization problem can be written as

$$\begin{aligned} \min_{\mathbf{x}} \quad & \sum_{e=1}^E \Phi_e(x^{(e)}) \\ \text{s.t.} \quad & \mathbf{Ax} = \mathbf{b} \end{aligned} \tag{13}$$

Our goal is to investigate Newton type methods for solving the problem in 13 in a distributed fashion. Before diving into these details, however, we next present basic ingredients needed for the remainder of the paper.

## 4.2 Dual Subgradient Method

The dual subgradient method optimizes the problem in Equation 13 by descending in the dual domain. The Lagrangian,  $l(\cdot) : \mathbb{R}^E \times \mathbb{R}^N \rightarrow \mathbb{R}$ , is given by

$$l(\mathbf{x}, \boldsymbol{\lambda}) = - \sum_{e=1}^E \Phi_e(x^{(e)}) + \boldsymbol{\lambda}^\top (\mathbf{Ax} - \mathbf{b}).$$

The dual function  $q(\boldsymbol{\lambda})$  is then derived as

$$\begin{aligned} q(\boldsymbol{\lambda}) &= \inf_{\mathbf{x} \in \mathbb{R}^E} l(\mathbf{x}, \boldsymbol{\lambda}) = \inf_{\mathbf{x} \in \mathbb{R}^E} \left( - \sum_{e=1}^E \Phi_e(x^{(e)}) + \boldsymbol{\lambda}^\top \mathbf{Ax} \right) - \boldsymbol{\lambda}^\top \mathbf{b} \\ &= \sum_{e=1}^E \inf_{x^{(e)} \in \mathbb{R}} \left( -\Phi_e(x^{(e)}) + (\boldsymbol{\lambda}^\top \mathbf{A})^{(e)} x^{(e)} \right) - \boldsymbol{\lambda}^\top \mathbf{b}. \end{aligned}$$

Hence, it can be clearly seen that the evaluation of the dual function  $q(\boldsymbol{\lambda})$  decomposes into  $E$  one-dimensional optimization problems. We assume that each of these optimization problems have an optimal solution, which is unique by the strict convexity of the functions  $\Phi_e$ . Denoting the solutions by  $x^{(e)}(\boldsymbol{\lambda})$  and using the first order optimality conditions, it can be seen that for each edge,  $e$ ,  $x^{(e)}(\boldsymbol{\lambda})$  is given by<sup>6</sup>

$$x^{(e)}(\boldsymbol{\lambda}) = [\dot{\Phi}_e]^{-1} \left( \lambda^{(i)} - \lambda^{(j)} \right), \tag{14}$$

where  $i \in \mathcal{N}$  and  $j \in \mathcal{N}$  denote the source and destining nodes of edge  $e = (i, j)$ , respectively (see [6] for details). Therefore, for an edge  $e$ , the evaluation of  $x^{(e)}(\boldsymbol{\lambda})$  can be performed based on local information about the edge's cost function and the dual variables of the incident nodes,  $i$  and  $j$ .

The dual problem is defined as  $\min_{\boldsymbol{\lambda} \in \mathbb{R}^N} q(\boldsymbol{\lambda})$ . Since the dual function is convex, the optimization problem can be solved using gradient descent according to

$$\boldsymbol{\lambda}_{k+1} = \boldsymbol{\lambda}_k - \alpha_k \mathbf{g}_k \text{ for all } k \geq 0, \tag{15}$$

with  $k$  being the iteration index, and  $\mathbf{g}_k = \mathbf{g}(\boldsymbol{\lambda}_k) = \nabla q(\boldsymbol{\lambda}_k)$  denoting the gradient of the dual function evaluated at  $\boldsymbol{\lambda} = \boldsymbol{\lambda}_k$ . Importantly, the computation of the gradient can be performed as  $\mathbf{g}_k = \mathbf{Ax}(\boldsymbol{\lambda}_k) - \mathbf{b}$ , with  $\mathbf{x}(\boldsymbol{\lambda}_k)$  being a vector composed of  $x^{(e)}(\boldsymbol{\lambda}_k)$  as determined by Equation 14.

<sup>6</sup>Note that if the dual is not continuously differentiable, the a generalized Hessian can be used.

Further, due to the sparsity pattern of the incidence matrix  $\mathbf{A}$ , the  $i^{th}$  element,  $g_k^{(i)}$ , of the gradient  $\mathbf{g}_k$  can be computed as

$$g_k^{(i)} = \sum_{e=(i,j)} x^{(e)}(\boldsymbol{\lambda}_k) - \sum_{e=(j,i)} x^{(e)}(\boldsymbol{\lambda}_k) - b^{(i)}. \quad (16)$$

Clearly, the algorithm in Equation 15 can be implemented in a distributed fashion, where each node,  $i$ , maintains information about its dual,  $\boldsymbol{\lambda}_k^{(i)}$ , and primal,  $x^{(e)}(\boldsymbol{\lambda}_k)$ , iterates of the outgoing edges  $e = (i, j)$ . Gradient components can then be evaluated as per 16 using only local information. Dual variables can then be updated using 15. Given the updated dual variables, the primal variables can be computed using 14.

Although the distributed implementation avoids the cost and fragility of collecting all information at centralized location, practical applicability of gradient descent is hindered by slow convergence rates. This motivates the consideration of Newton methods discussed next.

### 4.3 Newton's Method for Dual Descent

Newton's method is a descent algorithm along a scaled version of the gradient. Its iterates are typically given by

$$\boldsymbol{\lambda}_{k+1} = \boldsymbol{\lambda}_k + \alpha_k \mathbf{d}_k \quad \text{for all } k \geq 0, \quad (17)$$

with  $\mathbf{d}_k$  being the Newton direction at iteration  $k$ , and  $\alpha_k$  denoting the step size. The Newton direction satisfies

$$\mathbf{H}_k \mathbf{d}_k = -\mathbf{g}_k, \quad (18)$$

with  $\mathbf{H}_k = \mathbf{H}(\boldsymbol{\lambda}_k) = \nabla^2 q(\boldsymbol{\lambda}_k)$  being the Hessian of the dual function at the current iteration  $k$ .

#### 4.3.1 Properties of the Dual and Assumptions

Here, we detail some assumptions needed by our approach. We also derive essential Lemmas quantifying properties of the dual Hessian.

**Assumption 1.** *The graph,  $\mathcal{G}$ , is connected, non-bipartite and has algebraic connectivity lower bound by a constant  $\omega$ .*

**Assumption 2.** *The cost functions,  $\Phi_e(\cdot)$ , in Equation 13 are*

1. *twice continuously differentiable satisfying*

$$\gamma \leq \ddot{\Phi}_e(\cdot) \leq \Gamma,$$

*with  $\gamma$  and  $\Gamma$  are constants; and*

2. *Lipschitz Hessian invertible for all edges  $e \in \mathcal{E}$*

$$\left| \frac{1}{\ddot{\Phi}_e(\mathbf{x})} - \frac{1}{\ddot{\Phi}_e(\hat{\mathbf{x}})} \right| \leq \delta \|\mathbf{x} - \hat{\mathbf{x}}\|.$$

The following two lemmas [5, 6] quantify essential properties of the dual Hessian which we exploit through our algorithm to determine the approximate Newton direction.

**Lemma 14.** *The dual objective  $q(\boldsymbol{\lambda}) = \boldsymbol{\lambda}^\top (\mathbf{A}\mathbf{x}(\boldsymbol{\lambda}) - \mathbf{b}) - \sum_e \Phi_e(\mathbf{x}(\boldsymbol{\lambda}))$  abides by the following two properties [5, 6]:*

1. *The dual Hessian,  $\mathbf{H}(\boldsymbol{\lambda})$ , is a weighted Laplacian of  $\mathcal{G}$ :*

$$\mathbf{H}(\boldsymbol{\lambda}) = \nabla^2 q(\boldsymbol{\lambda}) = \mathbf{A} [\nabla^2 f(\mathbf{x}(\boldsymbol{\lambda}))]^{-1} \mathbf{A}^\top.$$

2. *The dual Hessian  $\mathbf{H}(\boldsymbol{\lambda})$  is Lipschitz continuous with respect to the Laplacian norm (i.e.,  $\|\cdot\|_{\mathcal{L}}$ ) where  $\mathcal{L}$  is the unweighted laplacian satisfying  $\mathcal{L} = \mathbf{A}\mathbf{A}^\top$  with  $\mathbf{A}$  being the incidence matrix of  $\mathcal{G}$ . Namely,  $\forall \boldsymbol{\lambda}, \bar{\boldsymbol{\lambda}}$ :*

$$\|\mathbf{H}(\bar{\boldsymbol{\lambda}}) - \mathbf{H}(\boldsymbol{\lambda})\|_{\mathcal{L}} \leq B \|\bar{\boldsymbol{\lambda}} - \boldsymbol{\lambda}\|_{\mathcal{L}},$$

*with  $B = \frac{\mu_n(\mathcal{L})\delta}{\gamma\sqrt{\mu_2(\mathcal{L})}}$  where  $\mu_n(\mathcal{L})$  and  $\mu_2(\mathcal{L})$  denote the largest and second smallest eigenvalues of the Laplacian  $\mathcal{L}$ .*

*Proof.* See Appendix. □

The following lemma follows from the above and is needed in the analysis later:

**Lemma 15.** *If the dual Hessian  $\mathbf{H}(\boldsymbol{\lambda})$  is Lipschitz continuous with respect to the Laplacian norm  $\|\cdot\|_{\mathcal{L}}$  (i.e., Lemma 14), then for any  $\boldsymbol{\lambda}$  and  $\hat{\boldsymbol{\lambda}}$  we have*

$$\|\nabla q(\hat{\boldsymbol{\lambda}}) - \nabla q(\boldsymbol{\lambda}) - \mathbf{H}(\boldsymbol{\lambda})(\hat{\boldsymbol{\lambda}} - \boldsymbol{\lambda})\|_{\mathcal{L}} \leq \frac{B}{2} \|\hat{\boldsymbol{\lambda}} - \boldsymbol{\lambda}\|_{\mathcal{L}}^2.$$

*Proof.* See Appendix. □

As detailed in [6], the exact computation of the inverse of the Hessian needed for determining the Newton direction can not be attained exactly in a distributed fashion. Authors in [5, 6] proposed approximation techniques for computing this direction. The effectiveness of these algorithms, however, highly depends on the accuracy of such an approximation. In this work, we propose a distributed approximator for the Newton direction capable of acquiring  $\epsilon$ -close solutions for any arbitrary  $\epsilon$ . Our results show that this new algorithm is capable of significantly surpassing others in literature where its performance accurately traces that of the standard centralized Newton approach.

#### 4.4 Accurate Distributed Newton Methods

Using the results of the distributed R-Hop solver, we propose a novel technique requiring only R-Hop communication for the distributed approximation of the Newton direction. Given the results of Lemma 14, we can determine the approximate Newton direction by solving a system of linear equations represented by an SDD matrix<sup>7</sup> with  $\mathbf{M}_0 = \mathbf{H}_k = \mathbf{H}(\boldsymbol{\lambda}_k)$ .

Formally, we consider the following iteration scheme:

$$\boldsymbol{\lambda}_{k+1} = \boldsymbol{\lambda}_k + \alpha_k \tilde{\mathbf{d}}_k, \quad (19)$$

with  $k$  representing the iteration number,  $\alpha_k$  the step-size, and  $\tilde{\mathbf{d}}_k$  denoting the approximate Newton direction. We determine  $\tilde{\mathbf{d}}_k$  by solving  $\mathbf{H}_k \tilde{\mathbf{d}}_k = -\mathbf{g}_k$  using Algorithm 3.2.2. It is easy to see that our approximation of the Newton direction,  $\tilde{\mathbf{d}}_k$ , satisfies

$$\|\tilde{\mathbf{d}}_k - \mathbf{d}_k\|_{\mathbf{H}_k} \leq \epsilon \|\mathbf{d}_k\|_{\mathbf{H}_k} \quad \text{with} \quad \tilde{\mathbf{d}}_k = -\mathbf{Z}_k \mathbf{g}_k,$$

where  $\mathbf{Z}_k$  approximates  $\mathbf{H}_k^\dagger$  according to the routine of Algorithm 3.2.2. The accuracy of this approximation is quantified in the following Lemma

**Lemma 16.** *Let  $\mathbf{H}_k = \mathbf{H}(\boldsymbol{\lambda}_k)$  be the Hessian of the dual function, then for any arbitrary  $\epsilon > 0$  we have*

$$e^{-\epsilon^2} \mathbf{v}^\top \mathbf{H}_k^\dagger \mathbf{v} \leq \mathbf{v}^\top \mathbf{Z}_k \mathbf{v} \leq e^{\epsilon^2} \mathbf{v}^\top \mathbf{H}_k^\dagger \mathbf{v}, \quad \forall \mathbf{v} \in \mathbf{1}^\perp.$$

*Proof.* See Appendix. □

##### 4.4.1 Convergence Guarantees

Given such an accurate approximation, next we analyze the iteration scheme of our proposed method showing that similar to standard Newton methods, we achieve superlinear convergence within a neighborhood of the optimal value. We start by analyzing the change in the Laplacian norm of the gradient between two successive iterations

**Lemma 17.** *Consider the following iteration scheme  $\boldsymbol{\lambda}_{k+1} = \boldsymbol{\lambda}_k + \alpha_k \tilde{\mathbf{d}}_k$  with  $\alpha_k \in (0, 1]$ , then, for any arbitrary  $\epsilon > 0$ , the Laplacian norm of the gradient,  $\|\mathbf{g}_{k+1}\|_{\mathcal{L}}$ , follows:*

$$\|\mathbf{g}_{k+1}\|_{\mathcal{L}} \leq \left[ 1 - \alpha_k + \alpha_k \epsilon \frac{\mu_n(\mathcal{L})}{\mu_2(\mathcal{L})} \sqrt{\frac{\Gamma}{\gamma}} \right] \|\mathbf{g}_k\|_{\mathcal{L}} + \frac{\alpha_k^2 B \Gamma^2 (1 + \epsilon)^2}{2 \mu_2^2(\mathcal{L})} \|\mathbf{g}_k\|_{\mathcal{L}}^2, \quad (20)$$

with  $\mu_n(\mathcal{L})$  and  $\mu_2(\mathcal{L})$  being the largest and second smallest eigenvalues of  $\mathcal{L}$ ,  $\Gamma$  and  $\gamma$  denoting the upper and lower bounds on the dual's Hessian, and  $B \in \mathbb{R}$  is defined in Lemma 15.

<sup>7</sup>For ease of presentation, we refrain some of the proofs to the appendix.

*Proof.* See Appendix.  $\square$

At this stage, we are ready to present the main results quantifying the convergence phases exhibited by our approach:

**Theorem 3.** Let  $\gamma, \Gamma, B$  be the constants defined in Assumption 2 and Lemma 14,  $\mu_n(\mathcal{L})$  and  $\mu_2(\mathcal{L})$  representing the largest and second smallest eigenvalues of the normalized laplacian  $\mathcal{L}$ ,  $\epsilon \in \left(0, \frac{\mu_2(\mathcal{L})}{\mu_n(\mathcal{L})} \sqrt{\frac{\Gamma}{\gamma}}\right)$  the precision parameter for the SDDM solver, and letting the optimal step-size parameter  $\alpha^* = \frac{e^{-\epsilon^2}}{(1+\epsilon)^2} \left(\frac{\gamma}{\Gamma} \frac{\mu_2(\mathcal{L})}{\mu_n(\mathcal{L})}\right)^2$ . Then the proposed algorithm given by the  $\lambda_{k+1} = \lambda_k + \alpha^* \tilde{d}_k$  exhibits the following three phases of convergence:

1. **Strict Decreases Phase:** While  $\|g_k\|_{\mathcal{L}} \geq \eta_1$ :

$$q(\lambda_{k+1}) - q(\lambda_k) \leq -\frac{1}{2} \frac{e^{-2\epsilon^2}}{(1+\epsilon)^2} \frac{\gamma^3}{\Gamma^2} \frac{\mu_2^2(\mathcal{L})}{\mu_n^4(\mathcal{L})} \eta_1^2.$$

2. **Quadratic Decrease Phase:** While  $\eta_0 \leq \|g_k\|_{\mathcal{L}} \leq \eta_1$ :

$$\|g_{k+1}\|_{\mathcal{L}} \leq \frac{1}{\eta_1} \|g_k\|_{\mathcal{L}}^2.$$

3. **Terminal Phase:** When  $\|g_k\|_{\mathcal{L}} \leq \eta_0$ :

$$\|g_{k+1}\|_{\mathcal{L}} \leq \sqrt{\left[1 - \alpha^* + \alpha^* \epsilon \frac{\mu_n(\mathcal{L})}{\mu_2(\mathcal{L})} \sqrt{\frac{\Gamma}{\gamma}}\right] \|g_k\|_{\mathcal{L}}},$$

where  $\eta_0 = \frac{\xi(1-\xi)}{\zeta}$  and  $\eta_1 = \frac{1-\xi}{\zeta}$ , with

$$\xi = \sqrt{\left[1 - \alpha^* + \alpha^* \epsilon \frac{\mu_n(\mathcal{L})}{\mu_2(\mathcal{L})} \sqrt{\frac{\Gamma}{\gamma}}\right]} \quad \text{with} \quad \zeta = \frac{B(\alpha^* \Gamma (1+\epsilon))^2}{2\mu_2^2(\mathcal{L})} \quad (21)$$

*Proof.* We will proof the above theorem by handling each of the cases separately. We start by considering the case when  $\|g_k\|_{\mathcal{L}} > \eta_1$  (i.e., **Strict Decrease Phase**). We have:

$$\begin{aligned} q(\lambda_{k+1}) &= q(\lambda_k) + g_k^\top (\lambda_{k+1} - \lambda_k) + \frac{1}{2} (\lambda_{k+1} - \lambda_k)^\top H(z) (\lambda_{k+1} - \lambda_k) \\ &= q(\lambda_k) + \alpha_k g_k^\top \tilde{d}_k + \frac{\alpha_k^2}{2} \tilde{d}_k^\top H(z) \tilde{d}_k \leq q(\lambda_k) + \alpha_k g_k^\top \tilde{d}_k + \frac{\alpha_k^2}{2\gamma} \tilde{d}_k^\top \mathcal{L} \tilde{d}_k, \end{aligned}$$

where the last steps holds since  $H(\cdot) \preceq \frac{1}{\gamma} \mathcal{L}$ . Noticing that  $\|\tilde{d}_k\|_{\mathcal{L}}^2 \leq \frac{\Gamma^2(1+\epsilon)^2}{\mu_2^2(\mathcal{L})} \|g_k\|_{\mathcal{L}}^2$  (see Appendix), the only remaining step needed is to evaluate  $g_k^\top \tilde{d}_k$ . Knowing that  $\tilde{d}_k = -Z_k g_k$ , we recognize

$$\begin{aligned} g_k^\top \tilde{d}_k &= -g_k^\top Z_k g_k \leq e^{-\epsilon^2} g_k^\top H_k^\dagger g_k \quad (\text{Lemma 16}) \\ &\leq -\frac{e^{-\epsilon^2}}{\mu_n(H_k)} g_k^\top g_k \leq -\frac{e^{-\epsilon}}{\mu_n(\mathcal{L})} g_k^\top g_k \leq -\frac{e^{-\epsilon^2} \gamma}{\mu_n(\mathcal{L})} \frac{g_k^\top \mathcal{L} g_k}{\mu_n(\mathcal{L})} = \frac{e^{-\epsilon^2} \gamma}{\mu_n^2(\mathcal{L})} \|g_k\|_{\mathcal{L}}^2, \end{aligned}$$

where the last step follows from the fact that  $\forall v \in \mathbb{R}^n : v^\top v \geq \frac{v^\top \mathcal{L} v}{\mu_n(\mathcal{L})}$ . Therefore, we can write

$$q(\lambda_{k+1}) - q(\lambda_k) \leq -\left[\alpha_k \frac{e^{-\epsilon^2} \gamma}{\mu_n^2(\mathcal{L})} - \alpha_k^2 \frac{\Gamma^2(1+\epsilon)^2}{2\gamma \mu_2^2(\mathcal{L})}\right] \|g_k\|_{\mathcal{L}}^2.$$

It is easy to see that  $\alpha_k = \alpha^* = \frac{e^{-\epsilon^2}}{(1+\epsilon)^2} \left(\frac{\gamma}{\Gamma} \frac{\mu_2(\mathcal{L})}{\mu_n(\mathcal{L})}\right)^2$  minimizes the right-hand-side of the above equation. Using  $\|g_k\|_{\mathcal{L}}$  gives the constant decrement in the dual function between two successive iterations as

$$q(\lambda_{k+1}) - q(\lambda_k) \leq -\frac{1}{2} \frac{e^{-2\epsilon^2}}{(1+\epsilon)^2} \frac{\gamma^3}{\Gamma^2} \frac{\mu_2^2(\mathcal{L})}{\mu_n^4(\mathcal{L})} \eta_1^2.$$

Considering the case when  $\eta_0 \leq \|g_k\|_{\mathcal{L}}^2 \eta_1$  (i.e., **Quadratic Decrease Phase**), Equation 20 can be rewritten as

$$\|g_{k+1}\|_{\mathcal{L}} \leq \xi^2 \|g_k\|_{\mathcal{L}} + \zeta \|g_k\|_{\mathcal{L}}^2,$$

with  $\xi$  and  $\zeta$  defined as in Equation 21. Further, noticing that since  $\|g_k\|_{\mathcal{L}} \geq \eta_0$  then  $\|g_k\|_{\mathcal{L}} \leq \frac{1}{\eta_0} \|g_k\|_{\mathcal{L}}^2 = \frac{\zeta}{\xi(1-\xi)} \|g_k\|_{\mathcal{L}}^2$ . Consequently the quadratic decrease phase is finalized by

$$\|g_{k+1}\|_{\mathcal{L}} \leq \zeta \left( \frac{\xi}{1-\xi} + 1 \right) \|g_k\|_{\mathcal{L}}^2 = \frac{\zeta}{1-\xi} \|g_k\|_{\mathcal{L}}^2 = \frac{1}{\eta_1} \|g_k\|_{\mathcal{L}}^2.$$

Finally, we handle the case where  $\|g_k\|_{\mathcal{L}} \leq \eta_0$  (i.e., **Terminal Phase**). Since  $\|g_k\|_{\mathcal{L}}^2 \leq \eta_0 \|g_k\|_{\mathcal{L}}$ , it is easy to see that

$$\begin{aligned} \|g_{k+1}\|_{\mathcal{L}} &\leq (\xi^2 + \zeta \eta_0) \|g_k\|_{\mathcal{L}} = (\xi^2 + \xi(1-\xi)) \|g_k\|_{\mathcal{L}} \\ &= \xi \|g_k\|_{\mathcal{L}} = \sqrt{\left[ 1 - \alpha^* + \alpha^* \epsilon \frac{\mu_n(\mathcal{L})}{\mu_2(\mathcal{L})} \sqrt{\frac{\Gamma}{\gamma}} \right]} \|g_k\|_{\mathcal{L}}. \end{aligned}$$

□

#### 4.4.2 Iteration Count and Message Complexity

Having proved the three convergence phases of our algorithm, we next analyze the number of iterations needed by each phase. These results are summarized in the following lemma:

**Lemma 18.** *Consider the algorithm given by the following iteration protocol:  $\lambda_{k+1} = \lambda_k + \alpha^* \tilde{d}_k$ . Let  $\lambda_0$  be the initial value of the dual variable, and  $q^*$  be the optimal value of the dual function. Then, the number of iterations needed by each of the three phases satisfy:*

1. The **strict decrease phase** requires the following number of iterations to achieve the quadratic phase:

$$N_1 \leq C_1 \frac{\mu_n(\mathcal{L})^2}{\mu_2^3(\mathcal{L})} \left[ 1 - \epsilon \frac{\mu_n(\mathcal{L})}{\mu_2(\mathcal{L})} \sqrt{\frac{\Gamma}{\gamma}} \right]^{-2},$$

$$\text{where } C_1 = C_1(\epsilon, \gamma, \Gamma, \delta, q(\lambda_0), q^*) = 2\delta^2(1+\epsilon)^2 [q(\lambda_0) - q^*] \frac{\Gamma^2}{\gamma}.$$

2. The **quadratic decrease phase** requires the following number of iterations to terminate:

$$N_2 = \log_2 \left[ \frac{\frac{1}{2} \log_2 \left( \left[ 1 - \alpha^* \left( 1 - \epsilon \frac{\mu_n(\mathcal{L})}{\mu_2(\mathcal{L})} \sqrt{\frac{\Gamma}{\gamma}} \right) \right] \right)}{\log_2(r)} \right],$$

where  $r = \frac{1}{\eta_1} \|g_{k'}\|_{\mathcal{L}}$ , with  $k'$  being the first iteration of the quadratic decrease phase.

3. The radius of the **terminal phase** is characterized by:

$$\rho_{\text{terminal}} \leq \frac{2 \left[ 1 - \epsilon \frac{\mu_n(\mathcal{L})}{\mu_2(\mathcal{L})} \sqrt{\frac{\Gamma}{\gamma}} \right]}{e^{-\epsilon^2 \gamma \delta}} \mu_n(\mathcal{L}) \sqrt{\mu_2(\mathcal{L})}.$$

*Proof.* See Appendix. □

Given the above result, the total message complexity can then be derived as:

$$\mathcal{O} \left( (N_1 + N_2) n \beta \left( \kappa(\mathbf{H}_k) \frac{1}{R} + R d_{\max} \right) \log \left( \frac{1}{\epsilon} \right) \right).$$

#### 4.4.3 Comparison to Existing Literature

Recent progress towards distributed second-order methods applied to network flow optimization adopt an approximation scheme based on the properties of the Hessian. Most of these methods (e.g., [7]) handle the simpler consensus setting and thus are not directly applicable to our more general setting. Closest to this work is that proposed in [5], where the authors approximate the Newton direction by truncating the Neumann expansion of the pseudo-inverse of the Hessian. This truncation, however, introduces additional error to the computation of the Newton direction leading to inaccurate results especially on large networks. The primary contrast to this work is that our method is capable of acquiring  $\epsilon$ -close (for any arbitrary  $\epsilon$ ) approximation to the Newton direction. In fact, the proposed method is almost identical to the exact Newton direction computed in a centralized manner (see Section 4.5).

Next, we formally derive the iteration counts for our method on four-special cases as benchmark comparisons. Since the main contribution (due to the presence of  $\log \log(\cdot)$  term in  $N_2$ ) in the iteration count is given by the strict decrease phase (see Lemma 18), we develop these results in terms of  $N_1$ . Also note that the corollaries provided below are substantially harder to acquire when compared to the standard Newton analysis due to the dependence of some constants, e.g.  $m$  and  $M$  on the graph structure. Opposed to the analysis performed by other methods, our analysis explicitly handles such dependencies leading to more realistic and accurate mathematical insights. This explains the relatively high dependency on  $n$  in some cases. Note, that in such case where these relations (i.e., parameter dependency on the graph structure) were not taken into account, substantial decrease in the complexity can be achieved at the compensate of accurate mathematical description.

##### Path Graph:

**Corollary 5.** *Given a path graph  $\mathcal{P}_n$  with  $n$  nodes, the strict-decrease phase of the distributed Newton method is given by:*

$$N_1(\mathcal{P}_n) = \mathcal{O} \left( n^6 \left[ 1 - \epsilon \frac{4n^2}{\pi^2} \sqrt{\frac{\Gamma}{\gamma}} \right]^{-2} \right)$$

##### Grid Graph:

**Corollary 6.** *For a grid graph  $\mathcal{G}_{k \times m}$  the strict-decrease phase of the distributed Newton method is given by:*

$$N_1(\mathcal{G}_{k \times m}) = \mathcal{O} \left( n^3 \left[ 1 - \epsilon \frac{8n^2}{\pi^2 (k^2 + \frac{n^2}{k^2})} \sqrt{\frac{\Gamma}{\gamma}} \right]^{-2} \right)$$

##### Scale-Free Graph:

**Corollary 7.** *For a scale free grid graph  $\mathcal{G}_{SN(n)}$  with  $n$ , the strict-decrease phase is given by:*

$$N_1(\mathcal{G}_{SN(n)}) = \mathcal{O} \left( n^4 \log n \left[ 1 - \epsilon \frac{n^{\frac{3}{2}} \log n}{4} \sqrt{\frac{\Gamma}{\gamma}} \right]^{-2} \right)$$

**d-Regular Ramanujan Expanders:** For such expanders the iteration count for the strict-decrease phase can be bounded by a constant.

#### 4.5 Experiments and Results

We evaluated the proposed distributed second-order method in three sets of experiments on randomly generated and Barbell networks. The goal was to assess the performance on networks exhibiting good and bad mixing times. We compared our algorithm's performance to: 1) exact-newton computed in a centralized fashion, 2) Accelerated Dual Descent (ADD) with two different splittings [5, 6], 3) dual sub-gradients, and 4) the fully distributed algorithms for convex optimization [41] (FDA). An  $\epsilon$  of  $\frac{1}{10,000}$ , a feasibility threshold of  $10^{-5}$ , and an R-Hop of 1 were provided to our SDDM solver for determining the approximate Newton direction. For all other methods free parameters were chosen as specified by the relevant papers. Feasibility and objective values were used as performance measures.

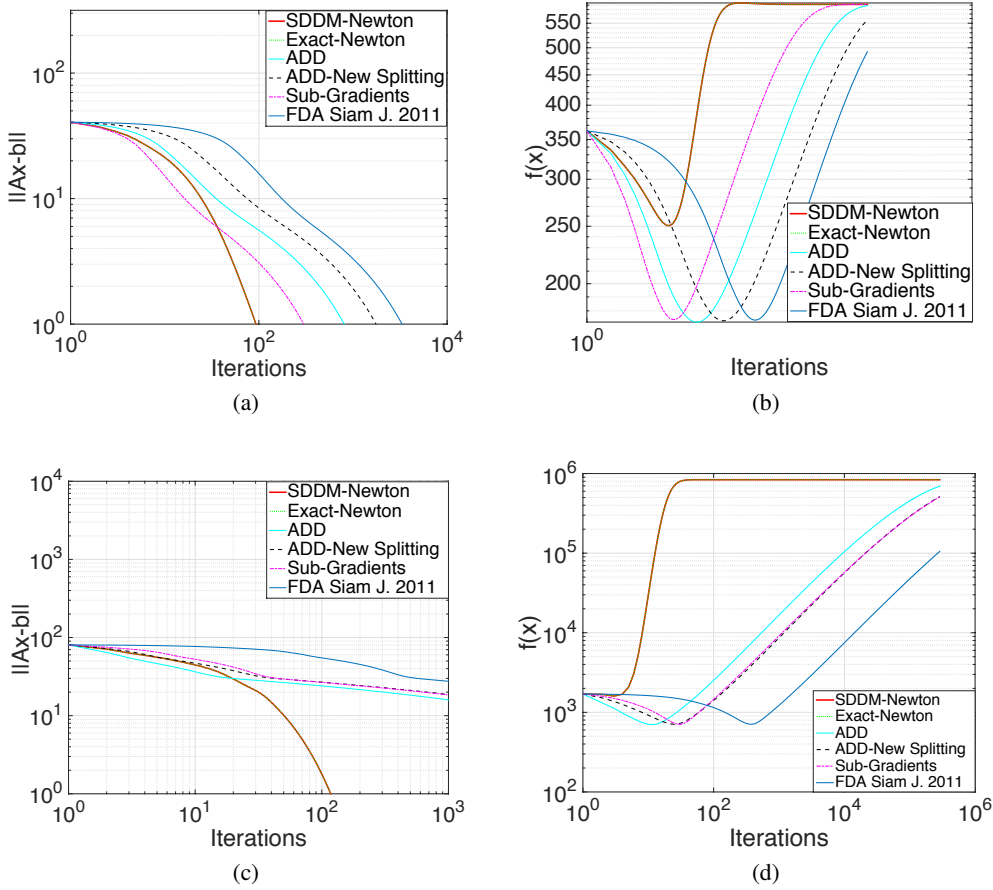


Figure 2: Performance metrics on two randomly generated networks, showing the primal objective,  $f(x_k)$ , and feasibility  $\|Ax_k - b\|$  as a function of the number of iterations  $k$  on a loglog scale. On a relatively small network (i.e., 20 nodes and 60 edges- Figures (a) and (b)) we outperform all method by an order of magnitude and trace the trajectory of exact Newton computed in a centralized fashion. On larger networks (i.e., 50 nodes and 150 edges-Figures (c) and (d)), SDDM-Newton is superior to all other algorithms, where it converges 5 orders of magnitude faster.

#### 4.5.1 Experiments & Results on Random Graphs

Two experiments on small (20 nodes, 60 edges) and large (50 nodes, 150 edges) random graphs were conducted. The random graphs were constructed in such a way that edges were drawn uniformly at random. The flow vectors,  $b$ , were chosen to place source and sink nodes  $\text{diam}(\mathcal{G})$  apart.

Results summarizing the primal objective value,  $f([x]^{(e)}) = \exp([x]^{(e)}) + \exp(-[x]^{(e)})$ , and feasibility,  $\|Ax_k - b\|$ , on both networks are shown in Figure 3. On small networks (i.e., 20 nodes and 60 edges), all algorithms perform relatively well. Clearly, our proposed approach (titled SDDM-Newton in the figures) outperforms, ADD, Sub-gradients, and the approach in [41] with about an order of magnitude. Another interesting realization is that SDDM-Newton accurately tracks the exact Newton method with its direction computed in a centralized fashion. The reason for such positive results, is that our algorithm is capable of approximating the Newton direction up-to-any arbitrary  $\epsilon > 0$  while abiding by the R-Hop constraint. For larger networks, Figures 2(c) and 2(d), SDDM-Newton is highly superior compared to other approaches. Here, our algorithm is capable of converging in about 3-5 orders of magnitude faster, i.e., we converge in about 3000 iterations as opposed to 6000 for ADD which showed the best performance among the other techniques. It is worth noting that we are again capable of tracing exact Newton due to the accuracy of our approximation.

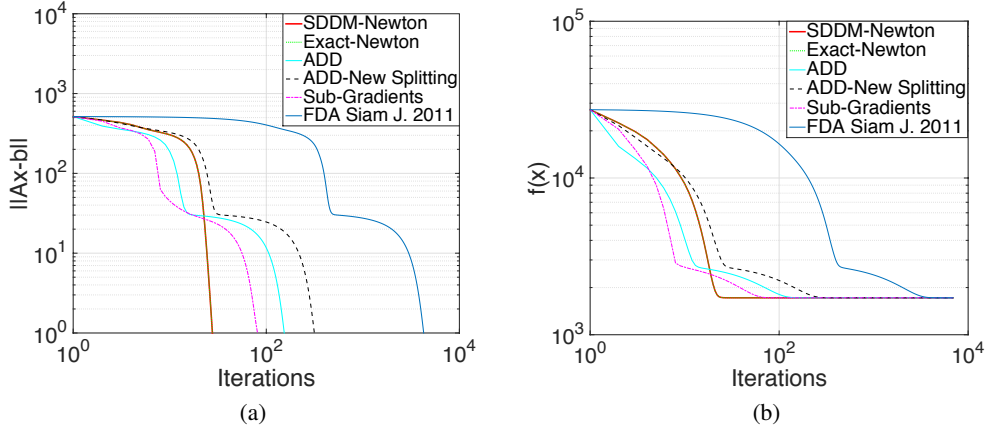


Figure 4: Feasibility and objective results on a 60 node bar graph. Again in these sets of experiments, our method is capable of outperforming all other methods. Furthermore, SDDM-Newton is again capable of tracing the exact Newton method.

#### 4.5.2 Experiments & Results on Bar Bell Graphs

In the second set of experiments we assessed the performance of SDDM-Newton on a barbell graph, Figure 3, of 60 nodes. The graph consisted of two 20 node cliques connected by a 20 nodes line graph is depicted in. We assign directed edges on the graph arbitrary and solve the network flow optimization problem with a cost of  $f([x]^{(e)}) = \exp([x]^{(e)}) + \exp(-[x]^{(e)})$ .

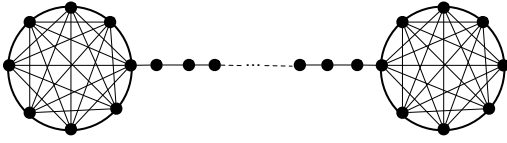


Figure 3: A high-level depiction of a barbell graph showing two cliques connected by a line graph.

Results in Figure 4(a) and 4(b) demonstrate the superiority of our algorithm to state-of-the-art methods. Here, again we are capable of converting faster than other techniques in about 2 magnitudes faster. It is worth noting that the second-best performing algorithm is ADD which is capable of converging in almost 3000 iterations.

in Figures 5(a) and 5(b), again validate the previous performance measures showing even better performance in both the objective value and feasibility.

Finally, we repeated the same experiments on a larger bar bell network formed of 120 nodes (two cliques with 40 nodes connected by a 40 node line graph). These results, demonstrated

#### 4.5.3 Measuring Message Complexity

Though successful, the experiments performed in the previous section show accuracy improvements without demonstrating per-iteration message complexities needed. To have a fair comparison to state-of-the-art methods, in this section we report such results on four different network topologies. Here, we show that our method requires a relatively slight increase in message complexity to trace the exact Newton direction.

These per-iteration values were determined by deriving bounds to each of the benchmark algorithms. For all methods except SDDM-Newton and that in [5], such complexity can be bounded by  $\mathcal{O}(d_{\max})$  with  $d_{\max}$  being the maximal degree. As for SDDM-Newton, the per-iteration complexity is upper-bounded by  $\mathcal{O}(\kappa(\mathcal{L}_{\mathcal{G}})d_{\max})$ , with  $\kappa(\mathcal{L}_{\mathcal{G}})$  being the condition number of the graph Laplacian. For the algorithm in [5] the message complexity satisfies  $\mathcal{O}(\kappa(\mathcal{L}_{\mathcal{G}})d_{\max} \log n)$  with  $n$  being the total number of nodes in  $\mathcal{G}$ . Immediately, we recognize that our algorithm is faster by a factor of  $\log n$  compared to that in [5]. Compared to other techniques, however, our method is slower by a factor of  $\kappa(\mathcal{L}_{\mathcal{G}})$ .

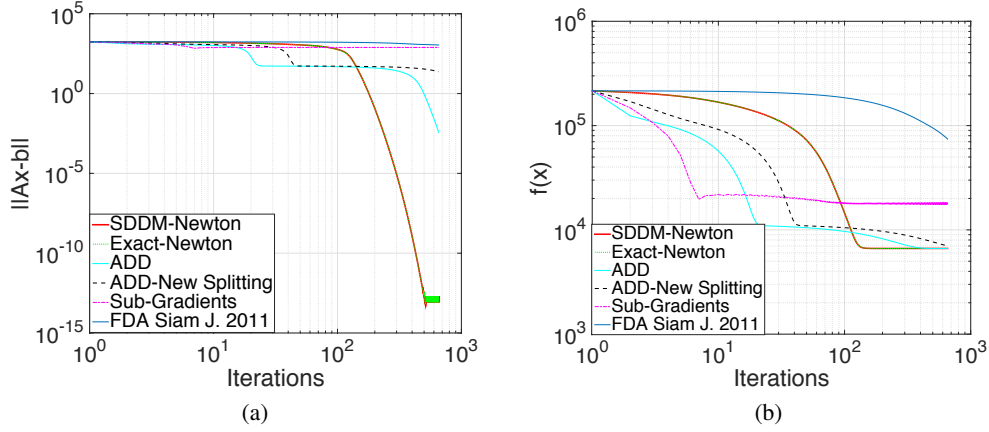


Figure 5: Feasibility and objective results on a 120 node bar graph. Again in these sets of experiments, our method is capable of outperforming all other methods. Furthermore, SDDM-Newton is again capable of tracing the exact Newton method.

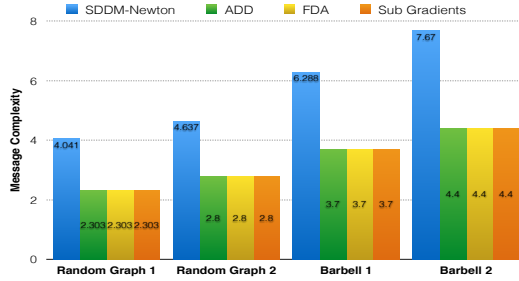


Figure 6: Message complexity results comparing our method to state-of-the-art techniques on four different network topologies. Graphs 1 and 2 correspond to random networks with 20 nodes, 60 edge and 100 nodes and 500 edge, respectively. Graphs 3 and 4 report message complexities needed for two barbell graphs.

To better quantify such a difference, we perform four sets of experiments on two randomly generated and two barbell networks with varying size. The random networks consisted of 20 nodes, 60 edges, and 120 nodes and 500 edges, respectively. Moreover, the barbell graphs followed the same construction of the previous section with sizes varying from 60 to 120 nodes. Comparison results showing the logarithm of the message complexity are reported in the bar graph of Figure 6. These demonstrate that SDDM-Newton requires a slight increase in the message complexity to trace the exact Newton direction.

## 5 Conclusion

In this paper we proposed a distributed solver for linear systems described by SDDM matrices. Our approach distributes that in [11] by

proposing the usage of an inverse approximated chain which can be computed in a distributed fashion. Precisely, two solvers were proposed. The first required full communication in the network, while the second restricts communication to the R-Hop neighborhood between the nodes.

We applied our solver to network flow optimization. This resulted in an efficient and accurate distributed second-order method capable of tracing exact Newton computed in a centralized fashion. We showed that similar to standard Newton, our methods are capable of achieving superlinear convergence in a neighborhood of the optimal solution. We extensively evaluated the proposed method on both randomly generated and barbell graphs. Results demonstrate that our method outperforms state-of-the-art techniques.

## References

- [1] S. Authuraliya and S. H. Low, *Optimization flow control with newton-like algorithm*, Telecommunications Systems **15** (200), 345-358.
- [2] D.P. Bertsekas, *Nonlinear programming*, Athena Scientific, Cambridge, Massachusetts, 1999.
- [3] D.P. Bertsekas, A. Nedic, and A.E. Ozdaglar, *Convex analysis and optimization*, Athena Scientific, Cambridge, Massachusetts, 2003.
- [4] S. Boyd and L. Vandenberghe, *Convex optimization*, Cambridge University Press, Cambridge, UK, 2004.
- [5] A. Jadbabaie, A. Ozdaglar, and M. Zargham, *A distributed newton method for network optimization*, Proceedings of IEEE CDC, 2009.
- [6] M. Zargham, A. Ribeiro, A. Ozdaglar, and A. Jadbabaie, *Accelerated Dual Descent for Network Optimization*, Proceedings of IEEE, 2011.
- [7] E. Wei, A. Ozdaglar, and A. Jadbabaie, *A distributed newton method for network utility maximization*, LIDS Technical Report 2823 (2010).
- [8] J. Sun and H. Kuo, *Applying a newton method to strictly convex separable network quadratic programs*, SIAM Journal of Optimization, 8, 1998.
- [9] R. Tyrrell Rockafellar, *Network Flows and Monotropic Optimization*, J. Wiley & Sons, Inc., 1984.
- [10] E. Gafni and D. P. Bertsekas, *Projected Newton Methods and Optimization of Multicommodity Flows*, IEEE Conference on Decision and Control (CDC), Orlando, Fla., Dec. 1982.
- [11] R. Peng, and D. A. Spielman, *An efficient parallel solver for SDD linear systems*, The 46th Annual ACM Symposium on Theory of Computing 2014.
- [12] A. Nedic and A. Ozdaglar, *Approximate primal solutions and rate analysis for dual subgradient methods*, SIAM Journal on Optimization, forthcoming (2008).
- [13] S. Low and D.E. Lapsley, *Optimization flow control, I: Basic algorithm and convergence*, IEEE/ACM Transactions on Networking **7** (1999), no. 6, 861-874.
- [14] A. Ribeiro and G. B. Giannakis, *Separation theorems of wireless networking*, IEEE Transactions on Information Theory (2007).
- [15] A. Ribeiro, *Ergodic stochastic optimization algorithms for wireless communication and networking*, IEEE Transactions on Signal Processing (2009).
- [16] O. Axelsson, *Iterative Solution Methods*, Cambridge University Press, New York, NY, USA, 1994.
- [17] D. P. Bertsekas and J. N. Tsitsiklis, *Parallel and Distributed Computation: Numerical Methods*, Prentice Hall, Inc., Upper Saddle River, NJ, USA, 1989.
- [18] E. G. Boman, B. Hendrickson, and S. A. Vavasis, *Solving elliptic finite element systems in near-linear time support preconditioners*, SIAM Journal on Numerical Analysis, 2008.
- [19] S. I. Daitch, and D. A. Spielman, *Faster Approximate Lossy Generalized Flow via Interior Point Algorithms*, In Proceedings of the 40<sup>th</sup> Annual ACM Symposium on Theory of Computing, 2008.
- [20] A. Madry, *Navigating Central Path with Electrical Flows: From Flows to Matching and Back*, In Proceedings of the 54<sup>th</sup> Annual IEEE Symposium On the Theory Of Computing (STOC), 2012.
- [21] X. Zhu, Z. Ghahramani, and J. D. Lafferty, *Semi-supervised Learning Using Gaussian Fields and Harmonic Functions*, In Proceedings of the 20<sup>th</sup> International Conference on Machine Learning, 2003.
- [22] D. Zhou, and B. Schoelkopf, *A Regularization Framework For Learning from Graph Data*, In the Statistical Relation Learning and Its Connections to Other Fields Workshop held at the International Conference on Machine Learning, 2004.
- [23] J.A. Kelner, and A. Madry, *Faster Generation of Random Spanning Trees*, In Foundations of Computer Science (FOCS), 2009.

- [24] D. A. Spielman, and S.-H. Teng, *Nearly-Linear Time Algorithm for Preconditioning and Solving Symmetric Diagonally Dominant Linear Systems*, CoPR, 2008.
- [25] A. Joshi, *Topics in Optimization and Sparse Linear Systems*, University of Illinois at Urbana-Champaign, 1996.
- [26] E. Boman, and B. Hendrickson, *Support Theory for Preconditioning*, SIAM Journal on Matrix Analysis and Applications, 2003.
- [27] D. A. Spielman, and S.-H. Teng, *Spectral Sparsification of Graphs*, CoPR, 2008.
- [28] I. Koutis, G. L. Miller, and R. Pend, *Approaching optimality for solving SDD systems*, CoPR, 2010.
- [29] I. Koutis, G. L. Miller, and R. Pend, *Solving SDD linear systems in time  $\tilde{O}(m \log n \log(1/\epsilon))$* , CoPR, 2011.
- [30] J. A. Kelner, and A. Madry, *Faster Generation of Random Spanning Trees*, CoPR, 2009.
- [31] I. Koutis, and G. L. Miller, *A Linear Work,  $O(n^{1/6})$  Time, Parallel Algorithm for Solving Planar Laplacians*, SODA, 2007.
- [32] G. Belloch, A. Gupta, I. Koutis, G. L. Miller, R. Peng, and K. Tandwongsan, *Near Linear-Work Parallel SDD Solvers, Low-Diameter Decomposition, and Low-Stretch Subgraphs*, CoPR, 2011.
- [33] J. Liu, S. Mou, and S. A. Morse, *An Asynchronous Distributed Algorithm for Solving a Linear Algebraic Equation*, In Proceedings of the 52<sup>nd</sup> Annual Conference on Decision and Control (CDC), 2013.
- [34] C. E. Lee, A. Ozdaglar, and D. Shah, *Solving Systems of Linear Equations: Locally and Asynchronously*, CoPR, 2014.
- [35] W. Casaca, G. Nonato, G. Taubin, *Laplacian Coordinates for Seeded Image Segmentation*, IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2014.
- [36] J. Nocedal, and S. J. Wright, *Numerical Optimization*, Springer, New York, 2006.
- [37] E.F. Kaasschieter, *Preconditioned Conjugate Gradients for Solving Singular Systems*, Journal of Computational and Applied Mathematics, 1988.
- [38] A. Olshevsky, *Linear Time Average Consensus on Fixed Graphs and Implications for Decentralized Optimization and Multi-Agent Control*, ArXiv e-prints, 2014.
- [39] Béla Bollobás, Oliver Riordan, Joel Spencer, and Gábor Tusnády, *The Degree Sequence of Scale-Free Random Graph Process*, Random Structures and Algorithms, 2001.
- [40] Rasul Tutunov, Haitham Bou-Ammar, Ali Jadbabaie, and Eric Eaton *On the Degree Distribution of Pólya Urn Graphical Processes*, ArXiv e-prints, 2014.
- [41] D. Mosk-Aoyama, T. Roughgarden, and D. Shah, *Fully Distributed Algorithms For Convex Optimization Problems*, Siam Journal on Optimization 2010.
- [42] C. Couprie, L. Grady, L. Najman, and H. Talbot, *Power Watershed: A Unifying Graph-Based Optimization Framework*, IEEE Transactions on Pattern Analysis and Machine Intelligence, 2011.
- [43] C. Rother, V. Kolmogorov, A. Blake, *GrabCut-Interactive Foreground Extraction using Iterated Graph Cuts*, ACM Transactions on Graphics (SIGGRAPH), 2004.
- [44] Y. Boykov, M.-P. Jolly, *Interactive Graph Cuts for Optimal Boundary Amp; Region Segmentation of Objects in N-D Images*, IEEE International Conference on Computer Vision, 2001.
- [45] L. Grady, *Random Walks for Image Segmentation*, IEEE Transactions on Pattern Analysis and Machine Intelligence, 2006.
- [46] J. Cousty, G. Bertrand, L. Najman, and M. Couprie, *Watershed Cuts: Minimum Spanning Forests and the Drop of Water Principle*, IEEE Transactions on Pattern Analysis and Machine Intelligence 2009.
- [47] O. Sorkine, *Differential Representations for Mesh Processing*, Computer Graphics Forum (Eurographics), 2006.
- [48] K. Xu, H. Zhang, D. Cohen-Or, and Y. Xiong, *Dynamic Harmonic Fields for Surface Processing*, IEEE International Conference on Shape Modelling and Applications, 2009.

- [49] F. Estrada, A. Jepson, *Benchmarking Image Segmentation Algorithms*, International Journal of Computer Vision, 2009.
- [50] P. Arbelaez, M. Maire, C. Fowlkes, J. Malik, *Contour Detection and Hierarchical Image Segmentation*, IEEE Transactions on Pattern Analysis and Machine Intelligence, 2011.
- [51] K. Gremban, *Combinatorial Preconditioners for Sparse, Symmetric, Diagonally Dominant Linear Systems*, PhD Thesis, Carnegie Mellon University, 1996.

## 6 SDDM Solver Proofs

In this appendix, we provide the complete and comprehensive proofs specific for the distributed SDDM solver.

**Lemma 19.** *Let  $\mathbf{Z}_0 \approx_\epsilon \mathbf{M}_0^{-1}$ , and  $\tilde{\mathbf{x}} = \mathbf{Z}_0 \mathbf{b}_0$ . Then  $\tilde{\mathbf{x}}$  is  $\sqrt{2^\epsilon(e^\epsilon - 1)}$  approximate solution of  $\mathbf{M}_0 \mathbf{x} = \mathbf{b}_0$ .*

*Proof.* Let  $\mathbf{x}^* \in \mathbb{R}^n$  be the solution of  $\mathbf{M}_0 \mathbf{x} = \mathbf{b}_0$ , then

$$\|\mathbf{x}^* - \tilde{\mathbf{x}}\|_{\mathbf{M}_0}^2 = (\mathbf{x}^* - \tilde{\mathbf{x}})^\top \mathbf{M}_0 (\mathbf{x}^* - \tilde{\mathbf{x}}) = (\mathbf{x}^*)^\top \mathbf{M}_0 \mathbf{x}^* + (\tilde{\mathbf{x}})^\top \mathbf{M}_0 \tilde{\mathbf{x}} - 2(\mathbf{x}^*)^\top \mathbf{M}_0 \tilde{\mathbf{x}} \quad (22)$$

Now, consider each term in (22) separately:

1.  $(\mathbf{x}^*)^\top \mathbf{M}_0 \tilde{\mathbf{x}} = \mathbf{b}_0^\top \mathbf{M}_0^{-1} \mathbf{M}_0 \mathbf{Z}_0 \mathbf{b}_0 = \mathbf{b}_0^\top \mathbf{Z}_0 \mathbf{b}_0$
2.  $(\mathbf{x}^*)^\top \mathbf{M}_0 \mathbf{x}^* = \mathbf{b}_0^\top \mathbf{M}_0^{-1} \mathbf{M}_0 \mathbf{M}_0^{-1} \mathbf{b}_0 = \mathbf{b}_0^\top \mathbf{M}_0^{-1} \mathbf{b}_0 \leq e^\epsilon \mathbf{b}_0^\top \mathbf{Z}_0 \mathbf{b}_0$
3.  $\tilde{\mathbf{x}}^\top \mathbf{M}_0 \tilde{\mathbf{x}} = \mathbf{b}_0^\top \mathbf{Z}_0 \mathbf{M}_0 \mathbf{Z}_0 \mathbf{b}_0 \leq e^\epsilon \mathbf{b}_0^\top \mathbf{Z}_0 \mathbf{b}_0$

Note that in the last step we used the fact that  $\mathbf{Z}_0 \approx_\epsilon \mathbf{M}_0^{-1}$  implies  $\mathbf{M}_0 \approx_\epsilon \mathbf{Z}_0^{-1}$ . Therefore, (22) can be rewritten as:

$$\|\mathbf{x}^* - \tilde{\mathbf{x}}\|_{\mathbf{M}_0}^2 \leq 2(e^\epsilon - 1) \mathbf{b}_0^\top \mathbf{Z}_0 \mathbf{b}_0 \quad (23)$$

Combining (23) with  $\mathbf{b}_0^\top \mathbf{Z}_0 \mathbf{b}_0 = (\mathbf{x}^*)^\top \mathbf{M}_0 \mathbf{Z}_0 \mathbf{M}_0 \mathbf{x}^* \leq e^\epsilon (\mathbf{x}^*)^\top \mathbf{M}_0 \mathbf{x}^*$ :

$$\|\mathbf{x}^* - \tilde{\mathbf{x}}\|_{\mathbf{M}_0}^2 \leq 2(e^\epsilon - 1) e^\epsilon (\mathbf{x}^*)^\top \mathbf{M}_0 \mathbf{x}^* = 2(e^\epsilon - 1) e^\epsilon \|\mathbf{x}^*\|_{\mathbf{M}_0}^2$$

□

**Lemma 20.** *Let  $\mathbf{M}_0 = \mathbf{D}_0 - \mathbf{A}_0$  be the standard splitting of  $\mathbf{M}_0$ . Let  $\mathbf{Z}'_0$  be the operator defined by  $\text{DistrRSolve}(\{[\mathbf{M}_0]_{k1}, \dots, [\mathbf{M}_0]_{kn}\}, [\mathbf{b}_0]_k, d)$  (i.e.,  $\mathbf{x}_0 = \mathbf{Z}'_0 \mathbf{b}_0$ ). Then*

$$\mathbf{Z}'_0 \approx_{\epsilon_d} \mathbf{M}_0^{-1}$$

Moreover, Algorithm 3 requires  $\mathcal{O}(dn^2)$  time steps.

*Proof.* The proof commences by showing that  $(\mathbf{D}_0^{-1} \mathbf{A}_0)^r$  and  $(\mathbf{A}_0 \mathbf{D}_0^{-1})^{-r}$  have a sparsity pattern corresponding to the r-hop neighborhood for any  $r \in \mathbb{N}$ . This case be shown using induction as follows

1. If  $r = 1$ , we have

$$[\mathbf{A}_0 \mathbf{D}_0^{-1}]_{ij} = \begin{cases} \frac{[\mathbf{A}_0]_{ij}}{[\mathbf{D}_0]_{ii}} & \text{if } j : \mathbf{v}_j \in \mathbb{N}_1(\mathbf{v}_i), \\ 0 & \text{otherwise.} \end{cases}$$

Therefore,  $\mathbf{A}_0 \mathbf{D}_0^{-1}$  has sparsity pattern corresponding to the 1-Hop neighborhood.

2. Assume that  $(\mathbf{A}_0 \mathbf{D}_0^{-1})^p$  has a sparsity patten corresponding to the p-hop neighborhood for all  $p : 1 \leq p \leq r - 1$ .

3. Now, consider  $(\mathbf{A}_0 \mathbf{D}_0^{-1})^r$ , where

$$[(\mathbf{A}_0 \mathbf{D}_0^{-1})^r]_{ij} = \sum_{k=1}^n [(\mathbf{A}_0 \mathbf{D}_0^{-1})^{r-1}]_{ik} [\mathbf{A}_0 \mathbf{D}_0^{-1}]_{kj} \quad (24)$$

Since  $\mathbf{A}_0 \mathbf{D}_0^{-1}$  is non negative then it is easy to see that  $[(\mathbf{A}_0 \mathbf{D}_0^{-1})^r]_{ij} \neq 0$  if and only if there exists  $k$  such that  $\mathbf{v}_k \in \mathbb{N}_{r-1}(\mathbf{v}_i)$  and  $\mathbf{v}_k \in \mathbb{N}_1(\mathbf{v}_j)$  (i.e.,  $\mathbf{v}_j \in \mathbb{N}_r(\mathbf{v}_i)$ ).

For  $\mathbf{D}_0^{-1} \mathbf{A}_0$ , the same results can be derived similarly.

Please notice that in **Part One** of DistrRSolve algorithm node  $\mathbf{v}_k$  computes (in a distributed fashion) the components  $[b_1]_k$  to  $[b_d]_k$  using the inverse approximated chain  $\mathcal{C} = \{\mathbf{A}_0, \mathbf{D}_0, \mathbf{A}_1, \mathbf{D}_1, \dots, \mathbf{A}_d, \mathbf{D}_d\}$ . Formally,

$$\mathbf{b}_i = \left[ \mathbf{I} + (\mathbf{A}_0 \mathbf{D}_0^{-1})^{2^{i-1}} \right] \mathbf{b}_{i-1} = \mathbf{b}_{i-1} + (\mathbf{A}_0 \mathbf{D}_0^{-1})^{2^{i-2}} \cdot (\mathbf{A}_0 \mathbf{D}_0^{-1})^{2^{i-2}} \mathbf{b}_{i-1}$$

Clearly, at the  $i^{th}$  iteration node  $\mathbf{v}_k$  requires the  $k^{th}$  row of  $(\mathbf{A}_0 \mathbf{D}_0^{-1})^{2^{i-2}}$  (i.e., the  $k^{th}$  row from the previous iteration) in addition to the  $j^{th}$  row of  $(\mathbf{A}_0 \mathbf{D}_0^{-1})^{2^{i-2}}$  from all nodes  $\mathbf{v}_j \in \mathbb{N}_{2^{i-1}}(\mathbf{v}_k)$  to compute the  $k^{th}$  row of  $(\mathbf{A}_0 \mathbf{D}_0^{-1})^{2^{i-1}}$ .

For computing  $\left[ (\mathbf{A}_0 \mathbf{D}_0^{-1})^{2^{i-1}} \right]_{kj}$ , node  $\mathbf{v}_k$  requires the  $k^{th}$  row and  $j^{th}$  column of  $(\mathbf{A}_0 \mathbf{D}_0^{-1})^{2^{i-2}}$ .

The problem, however, is that node  $\mathbf{v}_j$  can only send the  $j^{th}$  row of  $(\mathbf{A}_0 \mathbf{D}_0^{-1})^{2^{i-2}}$  which can be easily seen not to be symmetric. To overcome this issue, node  $\mathbf{v}_k$  has to compute the  $j^{th}$  column of  $(\mathbf{A}_0 \mathbf{D}_0^{-1})^{2^{i-2}}$  based on its  $j^{th}$  row. The fact that  $\mathbf{D}_0^{-1} (\mathbf{A}_0 \mathbf{D}_0^{-1})^{2^{i-2}}$  is symmetric, manifests that for  $r = 1, \dots, n$

$$\frac{\left[ (\mathbf{A}_0 \mathbf{D}_0^{-1})^{2^{i-2}} \right]_{rj}}{[\mathbf{D}_0]_{rr}} = \frac{\left[ (\mathbf{A}_0 \mathbf{D}_0^{-1})^{2^{i-2}} \right]_{jr}}{[\mathbf{D}_0]_{jj}}$$

Hence, for all  $r = 1, \dots, n$

$$\left[ (\mathbf{A}_0 \mathbf{D}_0^{-1})^{2^{i-2}} \right]_{rj} = \frac{[\mathbf{D}_0]_{rr}}{[\mathbf{D}_0]_{jj}} \left[ (\mathbf{A}_0 \mathbf{D}_0^{-1})^{2^{i-2}} \right]_{jr} \quad (25)$$

Now, let's analyze the time complexity of computing components  $[b_1]_k, [b_2]_k, \dots, [b_d]_k$ .

**Time Complexity Analysis:** At each iteration  $i$ , node  $\mathbf{v}_k$  receives the  $j^{th}$  row of  $(\mathbf{A}_0 \mathbf{D}_0^{-1})^{2^{i-2}}$  from all nodes  $\mathbf{v}_j \in \mathbb{N}_{2^{i-1}}(\mathbf{v}_k)$ . using Equation 25, node  $\mathbf{v}_k$  computes the corresponding columns as well as the product of these columns with the  $k^{th}$  row of  $(\mathbf{A}_0 \mathbf{D}_0^{-1})^{2^{i-2}}$ . Therefore, the time complexity at the  $i^{th}$  iteration is  $\mathcal{O}(n^2 + \text{diam}(\mathcal{G}))$ , where  $n^2$  is responsible for the  $k^{th}$  row computation, and  $\text{diam}(\mathcal{G})$  represents the communication cost between the nodes. Using the fact that  $\text{diam}(\mathcal{G}) \leq n$ , the total complexity of **Part One** in DistrRSolve algorithm is  $\mathcal{O}(dn^2)$ .

In **Part Two**, node  $\mathbf{v}_k$  computes (in a distributed fashion)  $[\tilde{x}_{d-1}]_k, [\tilde{x}_{d-2}]_k, \dots, [\tilde{x}_0]_k$  using the same inverse approximated chain  $\mathcal{C} = \{\mathbf{A}_0, \mathbf{D}_0, \mathbf{A}_1, \mathbf{D}_1, \dots, \mathbf{A}_d, \mathbf{D}_d\}$ .

$$\begin{aligned} \mathbf{x}_i &= \frac{1}{2} \mathbf{D}_0^{-1} \mathbf{b}_i + \frac{1}{2} \left[ \mathbf{I} + (\mathbf{D}_0^{-1} \mathbf{A}_0)^{2^i} \right] \mathbf{x}_{i+1} = \frac{1}{2} \mathbf{D}_0^{-1} \mathbf{b}_i + \frac{1}{2} \mathbf{x}_{i+1} \\ &\quad + \frac{1}{2} (\mathbf{D}_0^{-1} \mathbf{A}_0)^{2^{i-1}} (\mathbf{D}_0^{-1} \mathbf{A}_0)^{2^{i-1}} \mathbf{x}_{i+1} \end{aligned} \quad (26)$$

for  $i = d-1, \dots, 1$ . Thus,

$$\mathbf{x}_0 = \frac{1}{2} \mathbf{D}_0^{-1} \mathbf{b}_0 + \frac{\mathbf{x}_1}{2} + \frac{1}{2} (\mathbf{D}_0^{-1} \mathbf{A}_0) \mathbf{x}_1$$

Similar to the analysis of **Part One** of DistrRSolve algorithm the time complexity of **Part Two** as well as the time complexity of the whole algorithm is  $\mathcal{O}(dn^2)$ .

Finally, using Lemma 5 for the inverse approximated chain  $\mathcal{C} = \{\mathbf{A}_0, \mathbf{D}_0, \mathbf{A}_1, \mathbf{D}_1, \dots, \mathbf{A}_d, \mathbf{D}_d\}$  yields:

$$\mathbf{Z}'_0 \approx_{\epsilon_d} \mathbf{M}_0^{-1}.$$

□

**Lemma 21.** *Let  $\mathbf{M}_0 = \mathbf{D}_0 - \mathbf{A}_0$  be the standard splitting. Further, let  $\epsilon_d < \frac{1}{3} \ln 2$  in the nverse approximated chain  $\mathcal{C} = \{\mathbf{A}_0, \mathbf{D}_0, \mathbf{A}_1, \mathbf{D}_1, \dots, \mathbf{A}_d, \mathbf{D}_d\}$ . Then  $\text{DistrESolve}(\{[\mathbf{M}_0]_{k1}, \dots, [\mathbf{M}_0]_{kn}\}, [\mathbf{b}_0]_k, d, \epsilon)$  requires  $\mathcal{O}(\log \frac{1}{\epsilon})$  iterations to return the  $k^{\text{th}}$  component of the  $\epsilon$  close approximation for  $\mathbf{x}^*$ .*

*Proof.* Notice that iterations in DistrESolve corresponds to Preconditioned Richardson Iteration:

$$\mathbf{y}_t = [\mathbf{I} - \mathbf{Z}'_0 \mathbf{M}_0] \mathbf{y}_{t-1} + \mathbf{Z}_0 \mathbf{b}_0$$

where  $\mathbf{Z}'_0$  is the operator defined by DistrRSolve and  $\mathbf{y}_0 = \mathbf{0}$ . Therefore, from Lemma 7:

$$\mathbf{Z}'_0 \approx_{\epsilon_d} \mathbf{M}_0^{-1}$$

Finally, applying Lemma 6 gives that DistrESolve algorithm needs  $\mathcal{O}(\log \frac{1}{\epsilon})$  iterations to  $k^{\text{th}}$  component of the  $\epsilon$  approximated solution for  $\mathbf{x}^*$ . □

**Lemma 22.** *Let  $\mathbf{M}_0 = \mathbf{D}_0 - \mathbf{A}_0$  be the standard splitting. Further, let  $\epsilon_d < \frac{1}{3} \ln 2$  in the inverse approximated chain  $\mathcal{C} = \{\mathbf{A}_0, \mathbf{D}_0, \mathbf{A}_1, \mathbf{D}_1, \dots, \mathbf{A}_d, \mathbf{D}_d\}$ . Then,  $\text{DistrESolve}(\{[\mathbf{M}_0]_{k1}, \dots, [\mathbf{M}_0]_{kn}\}, [\mathbf{b}_0]_k, d, \epsilon)$  requires  $\mathcal{O}(dn^2 \log(\frac{1}{\epsilon}))$  time steps.*

*Proof.* Each iteration of DistrESolve algorithm calls DistrRSolve routine, therefore, using the above the total time complexity of f DistrESolve algorithm is  $\mathcal{O}(dn^2 \log(\frac{1}{\epsilon}))$  time steps □

**Lemma 23.** *Let  $\mathbf{M}_0 = \mathbf{D}_0 - \mathbf{A}_0$  be the standard splitting. Further, let  $\epsilon_d < \frac{1}{3} \ln 2$ . Then Algorithm 8 requires  $\mathcal{O}(\log \frac{1}{\epsilon})$  iterations to return the  $k^{\text{th}}$  component of the  $\epsilon$  close approximation to  $\mathbf{x}^*$ .*

*Proof.* Please note that the iterations of EDistRSolve correspond to a distributed version of the preconditioned Richardson iteration scheme

$$\mathbf{y}_t = [\mathbf{I} - \mathbf{Z}'_0 \mathbf{M}_0] \mathbf{y}_{t-1} + \mathbf{Z}'_0 \mathbf{b}_0$$

with  $\mathbf{y}_0 = \mathbf{0}$  and  $\mathbf{Z}'_0$  being the operator defined by RDistRSolve. From Lemma 3.8 it is clear that  $\mathbf{Z}'_0 \approx_{\epsilon_d} \mathbf{M}_0^{-1}$ . Applying Lemma 2.12, provides that EDistRSolve requires  $\mathcal{O}(\log \frac{1}{\epsilon})$  iterations to return the  $k^{\text{th}}$  component of the  $\epsilon$  close approximation to  $\mathbf{x}^*$ . Finally, since EDistRSolve uses procedure RDistRSolve as a subroutine, it follows that for each node  $v_k$  only communication between the R-hope neighbors is allowed. □

**Lemma 24.** *Let  $\mathbf{M}_0 = \mathbf{D}_0 - \mathbf{A}_0$  be the standard splitting and let  $\epsilon_d < \frac{1}{3} \ln 2$ , then EDistRSolve requires  $\mathcal{O}((2^d/R\alpha + \alpha R d_{\max}) \log(1/\epsilon))$  time steps. Moreover, for each node  $v_k$ , EDistRSolve only uses information from the R-Hop neighbors.*

*Proof.* Notice that at each iteration EDistRSolve calls RDistRSolve as a subroutine, therefore, for each node  $v_k$  only R-hop communication is allowed. Lemma 3.8 gives that the time complexity of each iteration is  $\mathcal{O}(\frac{2^d}{R}\alpha + \alpha R d_{\max})$ , and using Lemma 3.9 immediately gives that the time complexity of  $\mathcal{O}((2^d/R\alpha + \alpha R d_{\max}) \log(1/\epsilon))$ . □

## 7 Distributed Newton Lemmas

**Lemma 25.** *The dual objective  $q(\lambda) = \lambda^\top (A\mathbf{x}(\lambda) - b) - \sum_e \Phi_e(\mathbf{x}(\lambda))$  abides by the following two properties [6]:*

1. *The dual Hessian,  $\mathbf{H}(\lambda)$ , is a weighted Laplacian of  $\mathcal{G}$ :*

$$\mathbf{H}(\lambda) = \nabla^2 q(\lambda) = A [\nabla^2 f(\mathbf{x}(\lambda))]^{-1} A^\top$$

2. *The dual Hessian  $\mathbf{H}(\lambda)$  is Lipschitz continuous with respect to the Laplacian norm (i.e.,  $\|\cdot\|_{\mathcal{L}}$ ) where  $\mathcal{L}$  is the unweighted laplacian satisfying  $\mathcal{L} = AA^\top$  with  $A$  being the incidence matrix of  $\mathcal{G}$ . Namely,  $\forall \bar{\lambda}, \lambda$ :*

$$\|\mathbf{H}(\bar{\lambda}) - \mathbf{H}(\lambda)\|_{\mathcal{L}} \leq B \|\bar{\lambda} - \lambda\|_{\mathcal{L}}$$

with  $B = \frac{\mu_n(\mathcal{L})\delta}{\gamma\sqrt{\mu_2(\mathcal{L})}}$  where  $\mu_n(\mathcal{L})$  and  $\mu_2(\mathcal{L})$  denote the largest and second smallest eigenvalues of the Laplacian  $\mathcal{L}$ .

*Proof.* For the first part see Lemma 1 in [6]. So, let's prove the second part:  
Let's denote  $\mathbf{W}(\lambda) = [\nabla^2 f(\mathbf{x}(\lambda))]^{-1}$ , then:

$$\mathbf{H}(\bar{\lambda}) - \mathbf{H}(\lambda) = A[\mathbf{W}(\bar{\lambda}) - \mathbf{W}(\lambda)]A^\top \quad (27)$$

Using that  $\|\mathbf{S}\|_{\mathcal{L}} = \sup_{v \in \mathbf{1}^\perp} \frac{\|\mathbf{S}v\|_{\mathcal{L}}}{\|v\|_{\mathcal{L}}}$  let's fix some  $v \in \mathbf{1}^\perp$  and consider the expression  $\|A[\mathbf{W}(\bar{\lambda}) - \mathbf{W}(\lambda)]A^\top v\|_{\mathcal{L}}^2$ :

$$\begin{aligned} & \|A[\mathbf{W}(\bar{\lambda}) - \mathbf{W}(\lambda)]A^\top v\|_{\mathcal{L}}^2 = \\ & v^\top A[\mathbf{W}(\bar{\lambda}) - \mathbf{W}(\lambda)]A^\top \mathcal{L} A[\mathbf{W}(\bar{\lambda}) - \mathbf{W}(\lambda)]A^\top v =^1 \\ & v^\top A[\mathbf{W}(\bar{\lambda}) - \mathbf{W}(\lambda)](A^\top A)^2[\mathbf{W}(\bar{\lambda}) - \mathbf{W}(\lambda)]A^\top v \leq^2 \\ & \mu_n^2(\mathcal{L}) v^\top A[\mathbf{W}(\bar{\lambda}) - \mathbf{W}(\lambda)]^2 A^\top v \leq \\ & \mu_n^2(\mathcal{L}) \mu_n^2(\|\mathbf{W}(\bar{\lambda}) - \mathbf{W}(\lambda)\|) v^\top A A^\top v \\ & = \mu_n^2(\mathcal{L}) \mu_n^2(\|\mathbf{W}(\bar{\lambda}) - \mathbf{W}(\lambda)\|) \|v\|_{\mathcal{L}}^2 \end{aligned}$$

We used in step <sup>(1)</sup>  $\mathcal{L} = AA^\top$ , and in step <sup>(2)</sup> we used that  $\mu_n(A^\top A) = \mu_n(AA^\top) = \mu_n(\mathcal{L})$ . Therefore, we have:

$$\|A[\mathbf{W}(\bar{\lambda}) - \mathbf{W}(\lambda)]A^\top\|_{\mathcal{L}} \leq \mu_n(\mathcal{L}) \mu_n(\|\mathbf{W}(\bar{\lambda}) - \mathbf{W}(\lambda)\|) \quad (28)$$

Now, we upper bound the expression  $\mu_n(\|\mathbf{W}(\bar{\lambda}) - \mathbf{W}(\lambda)\|)$ :

$$\begin{aligned} \mu_n(\|\mathbf{W}(\bar{\lambda}) - \mathbf{W}(\lambda)\|) & \leq \max_{e \in \mathcal{E}} \left| \frac{1}{\ddot{\Phi}_e(\mathbf{x}_e(\bar{\lambda}))} - \frac{1}{\ddot{\Phi}_e(\mathbf{x}_e(\lambda))} \right| \leq \\ & \delta \max_{e \in \mathcal{E}} |\mathbf{x}_e(\bar{\lambda}) - \mathbf{x}_e(\lambda)| \end{aligned} \quad (29)$$

In the last transition we used Assumption 2. Now, using formulae for the derivative of the inverse function we have:

$$\left| \frac{\partial}{\partial \lambda_i} [\dot{\Phi}]^{-1}(\lambda) \right| = \frac{1}{\ddot{\Phi}([\dot{\Phi}]^{-1}(\lambda))} \leq \frac{1}{\gamma} \quad (30)$$

Hence,  $[\dot{\Phi}]^{-1}(\lambda)$  is bounded, and therefore  $[\dot{\Phi}]^{-1}(\lambda)$  is Lipschitz continuous with constant  $L' = \frac{1}{\gamma}$ .

Now, because  $\mathbf{x}_e(\lambda) = [\dot{\Phi}]^{-1}(\lambda_i - \lambda_j)$ , we have that  $\mathbf{x}_e(\lambda)$  is Lipschitz continuous with corresponding constant  $L'$ . Hence,  $\forall e \in \mathcal{E}$ :

$$|\mathbf{x}_e(\bar{\lambda}) - \mathbf{x}_e(\lambda)| \leq \frac{1}{\gamma} \|\bar{\lambda} - \lambda\|_2 \quad (31)$$

Now we are ready to prove the following

Claim: For all  $e \in \mathcal{E}$  and for any  $\bar{\lambda}, \lambda$ :

$$|\mathbf{x}_e(\bar{\lambda}) - \mathbf{x}_e(\lambda)| \leq \frac{1}{\gamma} \frac{\|\bar{\lambda} - \lambda\|_{\mathcal{L}}}{\sqrt{\mu_2(\mathcal{L})}} \quad (32)$$

*Proof.* Consider three cases:

1.  $\bar{\lambda} - \lambda \in \mathbf{1}^\perp$ . In this case, using that  $\forall v \in \mathbf{1}^\perp$  :  
 $v^\top v \leq \frac{v^\top \mathcal{L} v}{\mu_2(\mathcal{L})}$  in (31):

$$|x_e(\bar{\lambda}) - x_e(\lambda)| \leq \frac{1}{\gamma} \|\bar{\lambda} - \lambda\|_2 \leq \frac{1}{\gamma} \frac{\|\bar{\lambda} - \lambda\|_{\mathcal{L}}}{\sqrt{\mu_2(\mathcal{L})}}$$

2.  $\bar{\lambda} - \lambda \in \text{Span}\{\mathbf{1}\}$  In this case  $\|\bar{\lambda} - \lambda\|_{\mathcal{L}} = 0$ , and we have  $\bar{\lambda} = \lambda + \alpha \mathbf{1}$ , hence  $\bar{\lambda}_i - \bar{\lambda}_j = \lambda_i + \alpha - (\lambda_j + \alpha) = \lambda_i - \lambda_j$ , which gives:

$$x_e(\bar{\lambda}) = [\dot{\Phi}]^{-1}(\bar{\lambda}_i - \bar{\lambda}_j) = [\dot{\Phi}]^{-1}(\lambda_i - \lambda_j) = x_e(\lambda)$$

Therefore,

$$|x_e(\bar{\lambda}) - x_e(\lambda)| = 0 = \frac{1}{\gamma} \frac{\|\bar{\lambda} - \lambda\|_{\mathcal{L}}}{\sqrt{\mu_2(\mathcal{L})}}.$$

Consequently, (32) is valid.

3.  $\bar{\lambda} - \lambda = \mathbf{u}_1 + \mathbf{u}_2$ , where  $\mathbf{u}_1 \in \mathbf{1}^\perp$ ,  $\mathbf{u}_2 \in \text{Span}\{\mathbf{1}\}$ . In this case  $\|\bar{\lambda} - \lambda\|_{\mathcal{L}} = \|\mathbf{u}_1\|_{\mathcal{L}}$ , and  $\bar{\lambda}_i - \bar{\lambda}_j = \lambda_i - \lambda_j + u_1(i) - u_1(j)$ . Notice that the same expression for  $\bar{\lambda}_i - \bar{\lambda}_j$  will be in the case when  $\bar{\lambda} = \lambda + \mathbf{u}_1$ . Hence, using the first case which proves the claim:

$$\begin{aligned} |x_e(\bar{\lambda}) - x_e(\lambda)| &= |[\dot{\Phi}]^{-1}(\bar{\lambda}_i - \bar{\lambda}_j) - [\dot{\Phi}]^{-1}(\lambda_i - \lambda_j)| = \\ &|[\dot{\Phi}]^{-1}(\lambda_i - \lambda_j + u_1(i) - u_1(j)) - [\dot{\Phi}]^{-1}(\lambda_i - \lambda_j)| \leq \frac{1}{\gamma} \|\mathbf{u}_1\|_2 \leq \frac{1}{\gamma} \frac{\|\mathbf{u}_1\|_{\mathcal{L}}}{\sqrt{\mu_2(\mathcal{L})}} = \frac{1}{\gamma} \frac{\|\bar{\lambda} - \lambda\|_{\mathcal{L}}}{\sqrt{\mu_2(\mathcal{L})}} \end{aligned}$$

□

Combining the above claim with (29) gives:

$$\mu_n(|\mathbf{W}(\bar{\lambda}) - \mathbf{W}(\lambda)|) \leq \frac{\delta}{\gamma} \frac{\|\bar{\lambda} - \lambda\|_{\mathcal{L}}}{\sqrt{\mu_2(\mathcal{L})}} \quad (33)$$

Using (33) in (28) leads us to:

$$\|\mathbf{H}(\bar{\lambda}) - \mathbf{H}(\lambda)\|_{\mathcal{L}} \leq B \|\bar{\lambda} - \lambda\|_{\mathcal{L}},$$

where  $B = \frac{\mu_n(\mathcal{L})\delta}{\gamma\sqrt{\mu_2(\mathcal{L})}}$ . □

**Lemma 26.** *If the dual Hessian  $\mathbf{H}(\lambda)$  is Lipschitz continuous with respect to the Laplacian norm  $\|\cdot\|_{\mathcal{L}}$  (i.e., Lemma 7), then for any  $\lambda$  and  $\hat{\lambda}$  we have*

$$\|\nabla q(\hat{\lambda}) - \nabla q(\lambda) - \mathbf{H}(\lambda)(\hat{\lambda} - \lambda)\|_{\mathcal{L}} \leq \frac{B}{2} \|\hat{\lambda} - \lambda\|_{\mathcal{L}}^2.$$

*Proof.* We apply the result of Fundamental Theorem of Calculus for the gradient  $\nabla q$  which implies for any vectors  $\lambda$  and  $\hat{\lambda}$  in  $\mathbb{R}^n$  we can write

$$\nabla q(\hat{\lambda}) = \nabla q(\lambda) + \int_0^1 \mathbf{H}(\lambda + t(\hat{\lambda} - \lambda))(\hat{\lambda} - \lambda) dt, \quad (34)$$

We proceed by adding and subtracting  $\mathbf{H}(\lambda)(\hat{\lambda} - \lambda)$  to the integral in the right hand side of (34). It follows that

$$\begin{aligned} \nabla q(\hat{\lambda}) &= \nabla q(\lambda) + \\ &\int_0^1 [\mathbf{H}(\lambda + t(\hat{\lambda} - \lambda)) - \mathbf{H}(\lambda)] (\hat{\lambda} - \lambda) + \mathbf{H}(\lambda)(\hat{\lambda} - \lambda) dt. \end{aligned} \quad (35)$$

we can separate the integral in (35) into two integrals as

$$\begin{aligned}\nabla q(\hat{\lambda}) &= \nabla q(\lambda) + \int_0^1 \left[ \mathbf{H}(\lambda + t(\hat{\lambda} - \lambda)) - \mathbf{H}(\lambda) \right] (\hat{\lambda} - \lambda) dt \\ &+ \int_0^1 \mathbf{H}(\lambda)(\hat{\lambda} - \lambda) dt.\end{aligned}\quad (36)$$

The second integral in the right hand side of (36) does not depend on  $t$  and we can simplify the integral as  $\mathbf{H}(\lambda)(\hat{\lambda} - \lambda)$ . This simplification implies that we can rewrite (36) as

$$\begin{aligned}\nabla q(\hat{\lambda}) &= \nabla q(\lambda) + \mathbf{H}(\lambda)(\hat{\lambda} - \lambda) + \\ &\int_0^1 \left[ \mathbf{H}(\lambda + t(\hat{\lambda} - \lambda)) - \mathbf{H}(\lambda) \right] (\hat{\lambda} - \lambda) dt,\end{aligned}\quad (37)$$

By rearranging terms in (37) and taking the norm of both sides we obtain

$$\begin{aligned}\|\nabla q(\hat{\lambda}) - \nabla q(\lambda) - \mathbf{H}(\lambda)(\hat{\lambda} - \lambda)\|_{\mathcal{L}} &= \\ \left\| \int_0^1 \left[ \mathbf{H}(\lambda + t(\hat{\lambda} - \lambda)) - \mathbf{H}(\lambda) \right] (\hat{\lambda} - \lambda) dt \right\|_{\mathcal{L}} &\leq \\ \int_0^1 \left\| \left[ \mathbf{H}(\lambda + t(\hat{\lambda} - \lambda)) - \mathbf{H}(\lambda) \right] (\hat{\lambda} - \lambda) \right\|_{\mathcal{L}} dt\end{aligned}\quad (38)$$

Now we are ready to prove the following

Claim: Let  $\mathbf{H}(\lambda)$  be the Hessian of the dual function  $q(\lambda)$ . Then, for any  $v \in \mathbb{R}^n$ :

$$\|[\mathbf{H}(\bar{\lambda}) - \mathbf{H}(\lambda)] v\|_{\mathcal{L}} \leq \|\mathbf{H}(\bar{\lambda}) - \mathbf{H}(\lambda)\|_{\mathcal{L}} \|v\|_{\mathcal{L}} \quad (39)$$

*Proof.* Consider three cases:

1.  $v \in \mathbf{1}^\perp$ . In this case (39) follows immediately from the definition:  $\|S\|_{\mathcal{L}} = \sup_{v \in \mathbf{1}^\perp} \frac{\|Sv\|_{\mathcal{L}}}{\|v\|_{\mathcal{L}}}$ .
2.  $v \in \text{Span}\{\mathbf{1}\}$ . In this case  $\|v\|_{\mathcal{L}} = 0$  and  $[\mathbf{H}(\bar{\lambda}) - \mathbf{H}(\lambda)]v = \mathbf{H}(\bar{\lambda})v - \mathbf{H}(\lambda)v = \mathbf{0} - \mathbf{0} = \mathbf{0}$  (because  $\mathbf{H}(\cdot)\mathbf{1} = \mathbf{0}$ ). Hence, (39) is correct
3.  $v = u_1 + u_2$ , where  $u_1 \in \mathbf{1}^\perp, u_2 \in \text{Span}\{\mathbf{1}\}$ . In this case  $\|v\|_{\mathcal{L}} = \|u_1\|_{\mathcal{L}}$ , and
$$\begin{aligned}\|[\mathbf{H}(\bar{\lambda}) - \mathbf{H}(\lambda)] v\|_{\mathcal{L}} &= \|[\mathbf{H}(\bar{\lambda}) - \mathbf{H}(\lambda)] (u_1 + u_2)\|_{\mathcal{L}} = \\ \|[\mathbf{H}(\bar{\lambda}) - \mathbf{H}(\lambda)] u_1\|_{\mathcal{L}} &\stackrel{(1)}{\leq} \|\mathbf{H}(\bar{\lambda}) - \mathbf{H}(\lambda)\|_{\mathcal{L}} \|u_1\|_{\mathcal{L}} = \\ \|\mathbf{H}(\bar{\lambda}) - \mathbf{H}(\lambda)\|_{\mathcal{L}} \|v\|_{\mathcal{L}}\end{aligned}$$

where in step <sup>(1)</sup> we used the first case result.

This proves the claim

□

Applying the above claim to (38) gives:

$$\begin{aligned}\|\nabla q(\hat{\lambda}) - \nabla q(\lambda) - \mathbf{H}(\lambda)(\hat{\lambda} - \lambda)\|_{\mathcal{L}} &\leq \\ \int_0^1 \|\mathbf{H}(\lambda + t(\hat{\lambda} - \lambda)) - \mathbf{H}(\lambda)\|_{\mathcal{L}} t \|\hat{\lambda} - \lambda\|_{\mathcal{L}} dt &\leq^2 \\ \int_0^1 B \|\hat{\lambda} - \lambda\|_{\mathcal{L}} t \|\hat{\lambda} - \lambda\|_{\mathcal{L}} dt &= B \|\hat{\lambda} - \lambda\|_{\mathcal{L}}^2 \int_0^1 t dt \\ &= \frac{B}{2} \|\hat{\lambda} - \lambda\|_{\mathcal{L}}^2\end{aligned}$$

where in step <sup>(1)</sup> we used the fact that  $\mathbf{H}(\cdot)$  is Lipshitz continuous with respect to the laplacian norm  $\|\cdot\|_{\mathcal{L}}$ . □

**Lemma 27.** Let  $\mathbf{H}_k = \mathbf{H}(\lambda_k)$  be the Hessian of the dual function, then for any arbitrary  $\epsilon > 0$  we have

$$e^{-\epsilon^2} \mathbf{v}^\top \mathbf{H}_k^\dagger \mathbf{v} \leq \mathbf{v}^\top \mathbf{Z}_k \mathbf{v} \leq e^{\epsilon^2} \mathbf{v}^\top \mathbf{H}_k^\dagger \mathbf{v}, \quad \forall \mathbf{v} \in \mathbf{1}^\perp$$

*Proof.* Lets  $\{\mu_i^{(k)}\}_{i=1}^n$  be the collection of eigenvalues of  $\mathbf{H}_k$  and  $\{\mathbf{u}^{(k)}_i\}$  are corresponding eigenvectors. Then

$$\mathbf{H}_k = \sum_{i=2}^n \mu_i^{(k)} \mathbf{u}^{(k)}_i \mathbf{u}^{(k)\top}_i \quad \mathbf{H}_k^\dagger = \sum_{i=2}^n \frac{1}{\mu_i^{(k)}} \mathbf{u}^{(k)}_i \mathbf{u}^{(k)\top}_i \quad (40)$$

where we use  $\mu_1^{(k)} = 0$  and  $\mathbf{u}^{(k)}_1 = \mathbf{1}$ . Now lets fix some  $\delta > 0$  and consider the matrix  $\mathbf{H}_{k,\delta} = \sum_{i=2}^n \mu_i^{(k)} \mathbf{u}^{(k)}_i \mathbf{u}^{(k)\top}_i + \delta \mathbf{1} \mathbf{1}^\top = \mathbf{H}_k + \delta \mathbf{1} \mathbf{1}^\top$ . The corresponding linear system will have the form:

$$\mathbf{H}_{k,\delta} \mathbf{d}_k = -\mathbf{g}_k \quad (41)$$

and the operator  $\mathbf{Z}_{k,\delta}$  defined by by *EDistRSolve* routine for (41) satisfies:

$$e^{-\epsilon^2} \mathbf{v}^\top \mathbf{H}_{k,\delta}^{-1} \mathbf{v} \leq \mathbf{v}^\top \mathbf{Z}_{k,\delta} \mathbf{v} \leq e^{\epsilon^2} \mathbf{v}^\top \mathbf{H}_{k,\delta}^{-1} \mathbf{v}, \quad \forall \mathbf{v} \in \mathbb{R}^n \quad (42)$$

Notice that  $\mathbf{H}_{k,\delta}^{-1} = \sum_{i=2}^n \frac{1}{\mu_i^{(k)}} \mathbf{u}^{(k)}_i \mathbf{u}^{(k)\top}_i + \frac{1}{\delta} \mathbf{1} \mathbf{1}^\top = \mathbf{H}_k^\dagger + \frac{1}{\delta} \mathbf{1} \mathbf{1}^\top$ . Hence, taking  $\mathbf{v} \in \mathbf{1}^\perp$  in (42):

$$e^{-\epsilon^2} \mathbf{v}^\top \mathbf{H}_k^\dagger \mathbf{v} \leq \mathbf{v}^\top \mathbf{Z}_{k,\delta} \mathbf{v} \leq e^{\epsilon^2} \mathbf{v}^\top \mathbf{H}_k^\dagger \mathbf{v}, \quad \forall \mathbf{v} \in \mathbf{1}^\perp \quad (43)$$

The last step is to take the limit  $\delta \rightarrow 0$  in (43) and notice that  $\mathbf{Z}_{k,\delta} \rightarrow \mathbf{Z}_k$ :

$$e^{-\epsilon^2} \mathbf{v}^\top \mathbf{H}_k^\dagger \mathbf{v} \leq \mathbf{v}^\top \mathbf{Z}_k \mathbf{v} \leq e^{\epsilon^2} \mathbf{v}^\top \mathbf{H}_k^\dagger \mathbf{v}, \quad \forall \mathbf{v} \in \mathbf{1}^\perp$$

□

**Lemma 28.** Consider the following iteration scheme  $\lambda_{k+1} = \lambda_k + \alpha_k \tilde{\mathbf{d}}_k$  with  $\alpha_k \in (0, 1]$ , then, for any arbitrary  $\epsilon > 0$ , the Laplacian norm of the gradient,  $\|\mathbf{g}_{k+1}\|_{\mathcal{L}}$ , follows:

$$\|\mathbf{g}_{k+1}\|_{\mathcal{L}} \leq \left[ 1 - \alpha_k + \alpha_k \epsilon \frac{\mu_n(\mathcal{L})}{\mu_2(\mathcal{L})} \sqrt{\frac{\Gamma}{\gamma}} \right] \|\mathbf{g}_k\|_{\mathcal{L}} + \frac{\alpha_k^2 B \Gamma^2 (1 + \epsilon)^2}{2 \mu_2^2(\mathcal{L})} \|\mathbf{g}_k\|_{\mathcal{L}}^2 \quad (44)$$

with  $\mu_n(\mathcal{L})$  and  $\mu_2(\mathcal{L})$  being the largest and second smallest eigenvalues of  $\mathcal{L}$ ,  $\Gamma$  and  $\gamma$  are constants from Assumption 2, and  $B \in \mathbb{R}$  is defined previously.

*Proof.* Because the dual function  $q(\lambda)$  has Hessian which is Lipschitz continuous with respect to the laplacian norm  $\|\cdot\|_{\mathcal{L}}$ , we can write:

$$\|\mathbf{g}_{k+1} - \mathbf{g}_k - H_k(\lambda_{k+1} - \lambda_k)\|_{\mathcal{L}} \leq \frac{B}{2} \|\lambda_{k+1} - \lambda_k\|_{\mathcal{L}}^2 \quad (45)$$

Using  $\lambda_{k+1} = \lambda_k + \alpha_k \tilde{\mathbf{d}}_k$  can be rewritten as:

$$\|\mathbf{g}_{k+1} - \mathbf{g}_k - \alpha_k H_k \tilde{\mathbf{d}}_k\|_{\mathcal{L}} \leq \frac{\alpha_k^2 B}{2} \|\tilde{\mathbf{d}}_k\|_{\mathcal{L}}^2 \quad (46)$$

Therefore,

$$\|\tilde{\mathbf{d}}_k\|_{\mathcal{L}}^2 \leq \frac{\Gamma^2 (1 + \epsilon)^2}{\mu_2^2(\mathcal{L})} \|\mathbf{g}_k\|_{\mathcal{L}}^2 \quad (47)$$

Since  $\|\mathbf{g}_k + \alpha_k H_k \tilde{\mathbf{d}}_k\|_{\mathcal{L}}$ . Let  $\tilde{\mathbf{d}}_k = \mathbf{d}_k + \mathbf{c}_k$ , then:

$$\|\mathbf{c}_k\|_{H_k} \leq \epsilon \|\mathbf{d}_k\|_{H_k} \quad (48)$$

and

$$\begin{aligned} \|\mathbf{g}_k + \alpha_k H_k \tilde{\mathbf{d}}_k\|_{\mathcal{L}} &= \|\mathbf{g}_k + \alpha_k H_k (\mathbf{d}_k + \mathbf{c}_k)\|_{\mathcal{L}} = \\ \|\mathbf{g}_k - \alpha_k \mathbf{g}_k + \alpha_k H_k \mathbf{c}_k\|_{\mathcal{L}} &\leq (1 - \alpha_k) \|\mathbf{g}_k\|_{\mathcal{L}} + \alpha_k \|H_k \mathbf{c}_k\|_{\mathcal{L}} \end{aligned} \quad (49)$$

Therefore, we need to evaluate  $\|H_k c_k\|_{\mathcal{L}}$ :

$$\begin{aligned}
\|H_k c_k\|_{\mathcal{L}}^2 &= c_k^T H_k L H_k c_k \leq \mu_n(L) c_k^T H_k^2 c_k \\
&\leq \mu_n(\mathcal{L}) \mu_n(H_k) c_k^T H_k c_k \leq \mu_n(L) \frac{\mu_n(L)}{\gamma} \epsilon^2 \|d_k\|_{H_k}^2 = \\
&\frac{\mu_n^2(\mathcal{L}) \epsilon^2}{\gamma} d_k^T H_k d_k = \frac{\mu_n^2(\mathcal{L}) \epsilon^2}{\gamma} g_k^T H_k^\dagger H_k H_k^\dagger g_k = \\
&\frac{\mu_n^2(L) \epsilon^2}{\gamma} g_k^T H_k^\dagger g_k \leq 2 \frac{\mu_n^2(\mathcal{L}) \epsilon^2}{\gamma} \frac{\Gamma}{\mu_2^2(L)} \|g_k\|_{\mathcal{L}}^2 = \epsilon^2 \frac{\mu_n^2(L)}{\mu_2^2(\mathcal{L})} \frac{\Gamma}{\gamma} \|g_k\|_{\mathcal{L}}^2
\end{aligned}$$

Hence,

$$\|H_k c_k\|_{\mathcal{L}} \leq \epsilon \frac{\mu_n(\mathcal{L})}{\mu_2(\mathcal{L})} \sqrt{\frac{\Gamma}{\gamma}} \|g_k\|_{\mathcal{L}} \quad (50)$$

Combining the above gives:

$$\|g_k + \alpha_k H_k \tilde{d}_k\|_{\mathcal{L}} \leq \left[ 1 - \alpha_k + \alpha_k \epsilon \frac{\mu_n(\mathcal{L})}{\mu_2(\mathcal{L})} \sqrt{\frac{\Gamma}{\gamma}} \right] \|g_k\|_{\mathcal{L}} \quad (51)$$

Therefore, we have:

$$\begin{aligned}
\|g_{k+1}\| &\leq \|g_k + \alpha_k H_k \tilde{d}_k\|_{\mathcal{L}} + \frac{\alpha_k^2 B}{2} \|\tilde{d}_k\|_{\mathcal{L}}^2 \leq \\
&\left[ 1 - \alpha_k + \alpha_k \epsilon \frac{\mu_n(\mathcal{L})}{\mu_2(\mathcal{L})} \sqrt{\frac{\Gamma}{\gamma}} \right] \|g_k\|_{\mathcal{L}} + \frac{\alpha_k^2 B \Gamma^2 (1 + \epsilon)^2}{2 \mu_2^2(L)} \|g_k\|_{\mathcal{L}}^2
\end{aligned}$$

□

**Lemma 29.** Consider the algorithm given by the following iteration protocol:  $\lambda_{k+1} = \lambda_{k+1} + \alpha^* \tilde{d}_k$ . Let  $\lambda_0$  be the initial value of the dual variable, and  $q^*$  be the optimal value of the dual function. Then, the number of iterations needed by each of the three phases satisfy:

1. The **strict decrease phase** requires the following number iterations to achieve the quadratic phase:

$$N_1 \leq C_1 \frac{\mu_n(\mathcal{L})^2}{\mu_2^3(\mathcal{L})} \left[ 1 - \epsilon \frac{\mu_n(\mathcal{L})}{\mu_2(\mathcal{L})} \sqrt{\frac{\Gamma}{\gamma}} \right]^{-2},$$

$$\text{where } C_1 = C_1(\epsilon, \gamma, \Gamma, \delta, q(\lambda_0), q^*) = 2\delta^2(1 + \epsilon)^2 [q(\lambda_0) - q^*] \frac{\Gamma^2}{\gamma}.$$

2. The **quadratic decrease phase** requires the following number of iterations to terminate:

$$N_2 = \log_2 \left[ \frac{\frac{1}{2} \log_2 \left( \left[ 1 - \alpha^* \left( 1 - \epsilon \frac{\mu_n(\mathcal{L})}{\mu_2(\mathcal{L})} \sqrt{\frac{\Gamma}{\gamma}} \right) \right] \right)}{\log_2(r)} \right],$$

$$\text{where } r = \frac{1}{\eta_1} \|g_{k'}\|_{\mathcal{L}}, \text{ with } k' \text{ being the first iteration of the quadratic decrease phase.}$$

3. The radius of the **terminal phase** is characterized by:

$$\rho_{\text{terminal}} \leq \frac{2 \left[ 1 - \epsilon \frac{\mu_n(\mathcal{L})}{\mu_2(\mathcal{L})} \sqrt{\frac{\Gamma}{\gamma}} \right]}{e^{-\epsilon^2 \gamma \delta}} \mu_n(\mathcal{L}) \sqrt{\mu_2(\mathcal{L})}.$$

*Proof.* We will start with strict decrease phase. From Theorem 1:

$$q(\lambda_{k+1}) - q(\lambda_k) \leq -\frac{1}{2} \frac{e^{-2\epsilon^2}}{(1 + \epsilon)^2} \frac{\gamma^3}{\Gamma^2} \frac{\mu_2^2(L)}{\mu_n^4(L)} \eta_1^2 \quad (52)$$

where  $\eta_1 = \frac{1-\mathcal{A}}{\mathcal{B}}$ , with:

$$\mathcal{A} = \sqrt{\left[1 - \alpha^* + \alpha^* \epsilon \frac{\mu_n(L)}{\mu_2(L)} \sqrt{\frac{\Gamma}{\gamma}}\right]}; \quad \mathcal{B} = \frac{B(\alpha^* \Gamma(1+\epsilon))^2}{2\mu_2^2(L)}$$

Hence:

$$\begin{aligned} q(\lambda_{k+1}) - q(\lambda_k) &\leq -\frac{1}{2} \frac{e^{-2\epsilon^2}}{(1+\epsilon)^2} \frac{\gamma^3}{\Gamma^2} \frac{\mu_2^2(L)}{\mu_n^4(L)} \eta_1^2 \leq \\ &= -\frac{1}{2} \frac{e^{-2\epsilon^2}}{(1+\epsilon)^2} \frac{\gamma^3}{\Gamma^2} \frac{\mu_2^2(L)}{\mu_n^4(L)} \frac{(1-\mathcal{A})^2}{\mathcal{B}^2} = \\ &= -\frac{1}{2} \frac{e^{-2\epsilon^2}}{(1+\epsilon)^2} \frac{\gamma^3}{\Gamma^2} \frac{\mu_2^2(L)}{\mu_n^4(L)} \frac{(1-\mathcal{A})^2}{\left(\frac{B(\alpha^* \Gamma(1+\epsilon))^2}{2\mu_2^2(L)}\right)^2} = \\ &= -2 \frac{e^{-2\epsilon^2}}{(1+\epsilon)^6} \frac{\gamma^3}{\Gamma^6} \frac{\mu_2^6(L)}{\mu_n^4(L)} \frac{(1-\mathcal{A})^2}{B^2(\alpha^*)^4} = \\ &= -2 \frac{e^{-2\epsilon^2}}{(1+\epsilon)^6} \frac{\gamma^3}{\Gamma^6} \frac{\mu_2^6(L)}{\mu_n^4(L)} \frac{(1-\mathcal{A})^2}{B^2 \left( \left( \frac{e^{-\epsilon^2}}{(1+\epsilon)^2} \left( \frac{\gamma}{\Gamma} \frac{\mu_2(L)}{\mu_n(L)} \right)^2 \right)^4 \right)} = \\ &= -2e^{2\epsilon^2} (1+\epsilon)^2 \frac{\Gamma^2}{\gamma^5} \frac{\mu_n^4(L)}{\mu_2^2(L)} \frac{1}{B^2} (1-\mathcal{A})^2 = \\ &= -2e^{2\epsilon^2} (1+\epsilon)^2 \frac{\Gamma^2}{\gamma^5} \frac{\mu_n^4(L)}{\mu_2^2(L)} \frac{1}{\left( \frac{\mu_n(L)\xi}{\gamma\sqrt{\mu_2(L)}} \right)^2} (1-\mathcal{A})^2 = \\ &= -2e^{2\epsilon^2} (1+\epsilon)^2 \frac{\Gamma^2}{\gamma^3} \frac{\mu_n^2(L)}{\mu_2(L)} \frac{(1-\mathcal{A})^2}{\xi^2} \end{aligned} \tag{53}$$

Now, notice that:

$$1 - \mathcal{A} = \frac{1 - \mathcal{A}^2}{1 + \mathcal{A}}$$

Moreover, because  $0 \leq \mathcal{A} \leq 1$ , therefore:

$$\frac{1}{4}(1 - \mathcal{A}^2)^2 \leq (1 - \mathcal{A})^2 \leq (1 - \mathcal{A}^2)^2 \tag{54}$$

Therefore, we can write:

$$q(\lambda_{k+1}) - q(\lambda_k) \leq -2e^{2\epsilon^2}(1+\epsilon)^2 \frac{\Gamma^2 \mu_n^2(L)}{\gamma^3 \mu_2(L)} \frac{(1-\mathcal{A})^2}{\xi^2} \leq \quad (55)$$

$$\begin{aligned} & -2e^{2\epsilon^2}(1+\epsilon)^2 \frac{\Gamma^2 \mu_n^2(L)}{\gamma^3 \mu_2(L)} \frac{1}{\xi^2} \frac{(1-\mathcal{A}^2)^2}{4} = \\ & -\frac{1}{2}e^{2\epsilon^2}(1+\epsilon)^2 \frac{\Gamma^2 \mu_n^2(L)}{\gamma^3 \mu_2(L)} \frac{1}{\xi^2} (\alpha^*)^2 \left[ 1 - \epsilon \frac{\mu_n(L)}{\mu_2(L)} \sqrt{\frac{\Gamma}{\gamma}} \right]^2 = \\ & -\frac{1}{2}e^{2\epsilon^2}(1+\epsilon)^2 \frac{\Gamma^2 \mu_n^2(L)}{\gamma^3 \mu_2(L)} \frac{1}{\xi^2} \left( \frac{e^{-\epsilon^2}}{(1+\epsilon)^2} \left( \frac{\gamma \mu_2(L)}{\Gamma \mu_n(L)} \right)^2 \right)^2 \times \\ & \left[ 1 - \epsilon \frac{\mu_n(L)}{\mu_2(L)} \sqrt{\frac{\Gamma}{\gamma}} \right]^2 \\ & = -\frac{1}{2\xi^2} \frac{1}{(1+\epsilon)^2} \frac{\gamma}{\Gamma^2} \frac{\mu_2^3(L)}{\mu_n^2(L)} \left[ 1 - \epsilon \frac{\mu_n(L)}{\mu_2(L)} \sqrt{\frac{\Gamma}{\gamma}} \right]^2 = \\ & -\frac{1}{2\xi^2(1+\epsilon)^2} \frac{\gamma}{\Gamma^2} \frac{\mu_2^3(L)}{\mu_n^2(L)} \left[ 1 - \epsilon \frac{\mu_n(L)}{\mu_2(L)} \sqrt{\frac{\Gamma}{\gamma}} \right]^2 \end{aligned} \quad (56)$$

Denote

$$\delta^* = \frac{1}{2\xi^2(1+\epsilon)^2} \frac{\gamma}{\Gamma^2} \frac{\mu_2^3(L)}{\mu_n^2(L)} \left[ 1 - \epsilon \frac{\mu_n(L)}{\mu_2(L)} \sqrt{\frac{\Gamma}{\gamma}} \right]^2,$$

then from (55) we have:

$$q(\lambda_{k+1}) - q(\lambda_k) \leq -\delta^*$$

Hence, the number of iterations required by the algorithm for the strict decrease phase is upper-bounded by:

$$\begin{aligned} N_1 & \leq \frac{(q(\lambda_0) - q^*)}{\delta^*} = \\ & 2\xi^2(1+\epsilon)^2 [q(\lambda_0) - q^*] \frac{\Gamma^2 \mu_n^2(L)}{\gamma \mu_2^3(L)} \left[ 1 - \epsilon \frac{\mu_n(L)}{\mu_2(L)} \sqrt{\frac{\Gamma}{\gamma}} \right]^{-2} \end{aligned}$$

where  $q^*$  - optimal value of dual function.

Now, let's analyze the quadratic decrease phase. We have, for  $\eta_0 \leq \|g_k\|_L < \eta_1$ :

$$\|g_{k+1}\|_L \leq \frac{1}{\eta_1} \|g_k\|_L^2$$

Hence,

$$\frac{1}{\eta_1} \|g_{k+1}\|_L \leq \left( \frac{1}{\eta_1} \|g_k\|_L \right)^2 \quad (57)$$

Now, denote  $l$  be the first iteration when quadratic phase is achieved, i.e  $\|g_l\|_L < \eta_1$ , therefore, for  $k + l$  iteration:

$$\frac{1}{\eta_1} \|g_{k+l}\|_L \leq \left( \frac{1}{\eta_1} \|g_{k+l-1}\|_L \right)^2 \leq \dots \leq \left( \frac{1}{\eta_1} \|g_l\|_L \right)^{2^k} = r^{2^k}$$

where we use notation  $r = \frac{1}{\eta_1} \|g_l\|_L$ ,  $r \in (0, 1)$ . Hence, the number of iterations ADD-SDDM algorithm requires to reach terminal phase is given by the following condition:

$$\eta_1 r^{2^k} < \eta_0$$

which immediately gives:

$$k \geq \log_2 \left[ \frac{\log_2 \left( \frac{\eta_0}{\eta_1} \right)}{\log_2 r} \right] = \log_2 \left[ \frac{\log_2 \mathcal{A}}{\log_2 r} \right]$$

Hence,

$$N_2 = \log_2 \left[ \frac{\frac{1}{2} \log_2 \left( \left[ 1 - \alpha^* \left( 1 - \epsilon \frac{\mu_n(L)}{\mu_2(L)} \sqrt{\frac{\Gamma}{\gamma}} \right) \right] \right)}{\log_2 r} \right]$$

Finally lets consider the radius of the terminal phase:

$$\begin{aligned} \rho_{terminal} = \eta_0 &= \frac{\mathcal{A}(1 - \mathcal{A})}{\mathcal{B}} \leq \frac{1 - \mathcal{A}^2}{\mathcal{B}} = \frac{\alpha^* \left[ 1 - \epsilon \frac{\mu_n(L)}{\mu_2(L)} \sqrt{\frac{\Gamma}{\gamma}} \right]}{\frac{B(\alpha^* \Gamma(1+\epsilon))^2}{2\mu_2^2(L)}} = \\ &= \frac{2 \left[ 1 - \epsilon \frac{\mu_n(L)}{\mu_2(L)} \sqrt{\frac{\Gamma}{\gamma}} \right]}{e^{-\epsilon^2} \gamma \xi} \mu_n(L) \sqrt{\mu_2(L)} \end{aligned}$$

□